MASTER EN SCIENCES INFORMATIQUES À FINALITÉ SPÉCIALISÉE EN SOFTWARE ENGINEERING

Description et approche expérimentale de raisonnement hybride au sein du Web sémantique

Sirjacques, Justin

Award date:
2018

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2017–2018

Description et approche expérimentale de raisonnement hybride au sein du Web sémantique

Justin Sirjacques

Maîtres de stage : Michel Vanden Bossche, Maxime Van Assche

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Wim Vanhoof

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Le Web sémantique est en pleine expansion depuis plusieurs années. Les ontologies en son cœur sont de plus en plus amples et font appel au raisonnement logique de façon croissante. Ce travail a pour but d'analyser les différents types de raisonnement que sont les raisonnements *forward* et *backward* afin d'étudier les différents moyens de les disposer les uns par rapport aux autres pour obtenir un raisonnement hybride performant. Ce travail utilise les raisonneurs actuellement mis en œuvre au sein du Web sémantique en tant que base pour la recherche et l'expérimentation.

Mots-clés : *forward chaining*, *backward chaining*, raisonnement hybride, Web sémantique

Abstract

The Semantic Web has been in full expansion for several years. The ontologies at its heart are growing in scope and increasingly appeal to logical reasoning. This thesis aims to analyze the different types of reasoning that are forward and backward reasoning in order to study the different ways to arrange them in relation to obtain an efficient hybrid reasoning. This thesis uses the reasoners currently implemented within the Semantic Web as a basis for research and experimentation.

Keywords : forward chaining, backward chaining, hybrid reasonning, semantic Web

Préface

Ce mémoire est présenté en vue de l'obtention du grade de Master en sciences informatiques à l'Université de Namur. Le travail de recherche et de rédaction a été réfalisé sous la supervision de Monsieur Wim Vanhoof, Professeur à l'Université de Namur. Une grande partie des recherches a été effectuée lors d'un stage sous la direction de Messieurs Vanden Bossche-Marquette et Van Assche, respectivement cofondateur/CEO de ODASE Ontologies et CTO de ODASE Ontologies. Ce mémoire est une suite directe de ce stage et présente donc mes recherches, mes analyses, mes résultats ainsi que les problèmes auxquels j'ai été confrontés.

Mes premiers remerciements vont à mon promoteur Wim Vanhoof pour son aide précieuse et son implication dans la rédaction de ce mémoire.

Ensuite, je tiens à remercier mes maîtres de stage, Messieurs Vanden Bossche-Marquette et Van Assche sans lesquelles mon travail n'aurait pas été fructueux. Leurs connaissances et expérience dans le domaine de recherche de ce mémoire m'ont fortement aidé. De plus le cadre de travail que ces derniers m'ont offet pendant les quatres mois de mon stage est un souvenir exceptionnel.

Je tiens également à apporter une attention particulière à Madison qui n'a pas pu profiter de ma pleine présence pendant ces longs mois de recherches et de rédaction. Merci également à elle pour sa relecture attentive et méticuleuse.

Pour terminer, je remercie les membres de ma famille qui ont relu le présent mémoire avant sa remise.

Table des matières

Table des figures

Liste des tableaux

Listings

Première partie

Introduction

Chapitre 1

Introduction

Depuis plusieurs années, le Web sémantique est en pleine extension, les recherches dans ce domaine se multiplient et se divercifient expansivement. Le Web sémantique se base sur des technologies telles que RDF, XML ou encore OWL permettant de modéliser des ontologies. Ces ontologies ont pour but de modéliser un ensemble de connaissances dans un domaine donné. Elles sont de plus en plus utilisées et permettent le partage de données en ligne. Les ontologies sont, de plus, parfaitement adaptées au raisonnement logique.

Le raisonnement au sein du Web sémantique, tel est le sujet de ce mémoire. Plusieurs types de raisonnement existent chacun basé sur des représentations de données et de règles différentes. Les deux types principaux sont le raisonnement *forward* et le raisonnement *backward*, les maîtriser est essentiel à ce travail qui a pour but de les associer afin d'obtenir un nouveau type de raisonnement qu'est le raisonnement hybride.

Pour ce faire, nous avons divisé ce travail en deux parties principales. La première partie intitulée "État de l'art" a pour but de décrire le Web sémantique en profondeur en décrivant sa structure couche après couche. Cette partie contient, au sein de son second chapitre, une description détaillée des différents types de raisonnement nécessaire à la réalisation de ce travail.

La seconde partie principale nommée "Contribution personnelle" contient elle aussi deux chapitres, l'un décrivant le contexte de réalisation de ce travail et plus précisément le raisonneur *backward* autour duquel le raisonneur *forward* doit être développé. Le second chapitre détaille le fonctionnement de ce raisonneur *forward* et explore les différentes possibilités de combinaison de ces deux raisonneurs afin d'obtenir un raisonnement hybride efficace. Des tests de comparaison ont également été réalisés et sont exposés en fin de cette seconde section.

Une dernière partie nommée "Travaux futurs et conclusion" succède à ces deux parties principales. Celle-ci expose trois directions de recherche potentiellement enrichissantes pour le raisonnement hybride et son application. Enfin, le dernier chapitre conclut ce travail.

Deuxième partie

État de l'art

Chapitre 2

Web sémantique

Le Web sémantique a été inspiré par la vision de Tim Berners-Lee [1], fondateur et directeur du World Wide Web Consortium (W3C) [2], d'un Web plus flexible, intégré, automatique, auto-adaptatif et offrant une expérience plus interactive et plus riche aux utilisateurs. Le W3C a développé un ensemble de normes et d'outils pour soutenir cette vision. Après plusieurs années de recherche et développement, ceux-ci sont maintenant utilisables et ont un impact réel.

Le Web sémantique, les outils et normes utilisés pour ce mémoire sont décrits dans les sections suivantes. Dans un premier temps, le Web sémantique est défini, suivrons les définitions des ontologies, de RDF [3], de RDFS [4] et de OWL [5].

2.1 Le Web sémantique

Le Web sémantique fait référence à un vaste espace d'échange de ressources dans lequel les performances d'exploitation de grands volumes de données sont fortement accrues. Ces taux de performances élevés sont dus à la transition des tâches de recherche et d'analyse des utilisateurs vers les machines qui sont dès lors capables d'accéder au contenu et de raisonner sur celui-ci [6].

2.1.1 Formalisme

Le Web actuel offre une quantité d'information immodérée dont une grande partie est redondante et uniquement compréhensible par l'Homme. Le Web sémantique a pour but de formaliser l'information afin de la rendre accessible et compréhensible par les machines. La formalisation est en effet la clé du Web sémantique. Une formalisation des connaissances dans une infrastructure bien définie est l'essence même du Web sémantique. Ce formalisme renforce l'accessibilité du Web "classique" tout en permettant la localisation, l'identification et la transformation de ressources de manière robuste et fiable. L'infrastructure doit pouvoir, d'une part, automatiser l'interopérabilité entre différents formalismes et d'autre part, valoriser et faciliter des raisonnements complexes tout en proposant un taux de validité élevé [1].

Cependant, le Web sémantique ne peut être réduit à cette infrastructure, une telle réduction serait limitative. En effet, les différentes applications développées sur cette infrastructure le représentent et en sont la preuve du concept. L'implémentation de ces applications est évidemment facilitée par les outils proposés par W3C.

2.1.2 Approche par couches

Le Web sémantique et son développement peuvent être divisés en plusieurs étapes, chaque étape consiste en la construction d'une couche supérieure à une autre. L'ensemble de ces couches est représenté dans la Figure 2.1 ci-dessous.

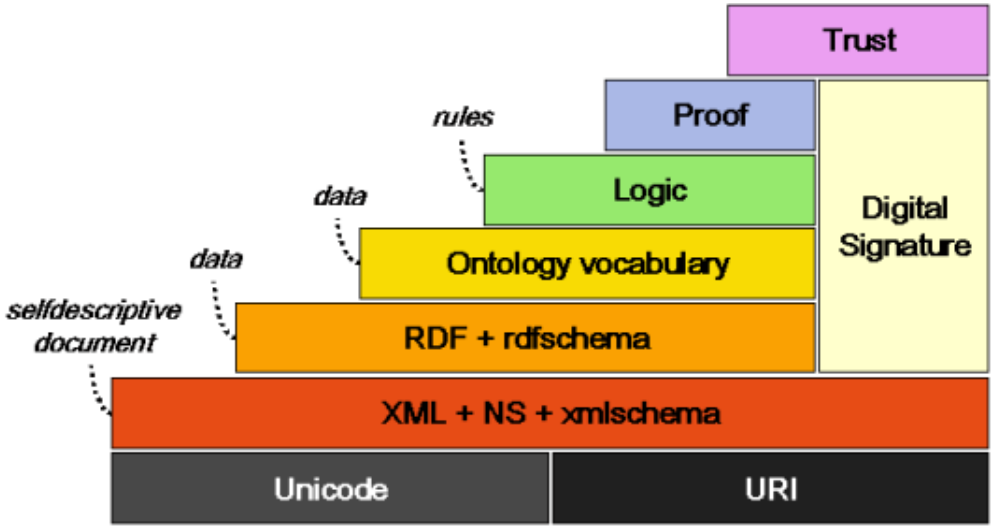

FIGURE 2.1 – Les couches du Web sémantique
Source: [7], p.19

Cette approche par couche est justifiée pragmatiquement par le fait qu'il est plus aisé de parvenir à un consensus sur de petites étapes plutôt que sur des innovations trop ambitieuses [7].

La Figure 2.1 décrit la vision par couche du Web sémantique, en voici une description de bas en haut :

— XML [8] : Ce langage qui compose la première couche du Web sémantique permet aux utilisateurs d'écrire des documents Web structurés suivant un vocabulaire pré-défini par leur soin. XML est un langage adapté pour le partage de documents via le Web.

— RDF [3] et RDF Schema (RDFS) [4] : RDF est un modèle de données permettant l'écriture et la représentation de déclaration en rapport avec des objets. RDFS, quant à lui, permet la définition de structures primitives afin d'organiser hiérarchiquement les objets. Bien que RDF ne dépende pas directement de XML, il utilise une syntaxe basée sur celui-ci. RDFS

est lui basé sur RDF. Il est donc naturel que ces deux technologies représentent une couche supérieure à XML. RDF et RDFS sont abordés de manière détaillée dans leurs sections, à savoir les Sections 2.3 et 2.4.

— Ontologie : Bien que RDFS puisse être vu comme un langage pour écrire des ontologies primitives, il ne permet pas la représentation de relations de complexité élevée entre objets. Une nouvelle couche est donc nécessaire pour la représentation détaillée d'ontologies. De façon similaire à RDF et RDFS, une section est dédiée aux ontologies, la Section 2.2.

— Logique : Cette couche ne concerne plus la représentation de données Web mais permet l'écriture de connaissances spécifiques à des ontologies qui permettront le raisonnement. Cette couche logique ainsi que ses technologies principales et les différents types de raisonnement sont abordés dans le Chapitre 3.

— Preuve : Cette couche consiste en un processus de déduction basé sur les couches inférieures. On parle donc de raisonnement sur les données via les règles. Le Chapitre 3 y est dédié.

— Vérité : Enfin, la couche confiance sera assurée par l'utilisation de signatures numériques et d'autres types de connaissances, sur la base des recommandations d'agents de confiance. La confiance est un concept crucial de haut niveau : le Web sémantique n'est abouti que si l'ensemble des utilisateurs ont confiance dans ses opérations et dans la véracité des informations qu'il fournit.

En 2004, dans leur livre [7], Grigoris Anthonion et Frank Van Harmelen affirmaient qu'en construisant le Web sémantique couche après couche, deux principes fondamentaux doivent être satisfaits :

— La compatibilité descendante : ce terme signifie que les individus maîtrisant une couche doivent être capables d'utiliser et d'interpréter les couches inférieures [1].

— La compréhension partielle ascendante : la conception doit être telle que les agents pleinement conscients d'une couche doivent être en mesure de tirer parti, au moins partiellement, des informations de niveaux supérieurs [2]. Évidemment, tous les outils d'implémentation ne doivent pas respecter cette fonctionnalité.

2.2 Ontologie

Le terme ontologie a, dans le passé, été limité à la sphère philosophique. En effet il correspond à la partie de la philosophie qui a pour objet l'étude des propriétés les plus

1. Par exemple : un agent connaissant le vocabulaire relatif aux ontologies est capable de mettre à profit toute information décrite dans RDF et RDFs. Voir Figure 2.1

2. Par exemple : un agent connaissant uniquement les sémantiques RDF et RDFS peut interpréter partiellement des connaissances décrites selon la sémantique propre aux ontologies. Voir Figure 2.1

générales de l'être, telles que l'existence, la possibilité, la durée, le devenir [3]. Dans le contexte informatique actuel, une ontologie représente un ensemble de connaissances propre à un domaine et facilite le partage de ces connaissances grâce un à vocabulaire précis et commun. La tendance principalement philosophique a été renversée, la communauté informatique investit dans la recherche ontologique de façon croissante. Les ontologies jouent un rôle spécifique dans une multitude de domaines de recherche, en voici une liste non-exhaustive dans lesquels l'importance des ontologies a été reconnue :

— L'ingénierie des connaissances [9] ;
— La gestion et organisation des connaissances [10] ;
— La conception de bases de données [11] ;
— L'intégration de l'information [12] ;
— La traduction en langage naturel [13].

Les ontologies représentent, dans un système d'information, des modèles de données d'un domaine précis ainsi que les relations entre ces données. La principale utilisation des ontologies est le raisonnement sur les objets du domaine concerné. L'expression suivante permet, grâce à une analogie, de comprendre le but d'une ontologie par rapport aux données.

> *« L'ontologie est aux données ce que la grammaire est au langage. »*
>
> Wikipedia

Une ontologie peut être visualisée comme un graphe représentant des concepts liés les uns aux autres par des relations sémantiques ou de subsomption [4].

Une ontologie est généralement composée d' :

— individus qui représentent des objets ou des instances ;
— classes, équivalentes à des ensembles, des collections, des concepts, des classes ou des types d'objets ;
— attributs qui sont des propriétés, des fonctionnalité, des caractéristiques ou encore des paramètres que peuvent avoir les classes ;
— relations de type sémantique ou de subsomption.

2.2.1 Types d'ontologie

En 2001 dans [14], Dieter Fensel différenciait les ontologies selon leur type, ce dernier en a identifié cinq :

3. Définition extraite de http ://www.cnrtl.fr/definition/ontologie
4. La subsomption est une inclusion entre des concepts. Par exemple : le concept de légume subsume celui de carotte.

— Les ontologies de domaine : elles capturent les connaissance d'un domaine particulier [5].

— Les ontologies de méta-donnée : elles fournissent un vocabulaire pour décrire des sources d'information en ligne.

— Les ontologies génériques : elles ont pour but de capturer des connaissances générales sur le monde en fournissant des notions et concepts de base.

— Les ontologies de représentation : elles ne sont pas propres à un domaine particulier, ces ontologies fournissent des entités de représentation sans spécifier ce qui devrait être représenté [6].

— Les autres types d'ontologie sont appelés les ontologies de tâches : elles fournissent des termes spécifiques pour des tâches particulières [7].

2.3 RDF

RDF ("Resource Description Framework") [3] est un langage universel développé dans le but de représenter et d'exploiter des méta-données. RDF peut être interprété comme le moyen de voir le Web comme un ensemble de ressources liées sémantiquement les unes aux autres.

2.3.1 Concepts fondamentaux

RDF se base sur l'utilisation de triples ($ressource - attribut - valeur$). Ces triples, pouvant être interprétés comme ($sujet - prédicat - objet$), sont à la base de la simplicité de ce modèle qui fût critiquée par certains alors que pour d'autres, elle est une des clés de l'utilisation et de l'acceptation de RDF.

Les trois concepts fondamentaux de RDF sont les ressources, les propriétés et les déclarations [7].

2.3.1.1 Ressource

Une ressource peut être vu comme un objet ou encore une "chose" dont nous souhaitons parler. Chaque ressource possède un URI qui permet de l'identifier, les URI sont abordés dans la Section 2.3.2 qui leur est consacrée.

2.3.1.2 Propriété

Les propriétés sont un type particulier de ressources. Elles décrivent les relations entre les ressources et sont également identifiées par des URI.

5. Par exemple : le domaine médical

6. Une ontologie représentation bien connue est l'ontologie des cadres qui définit des concepts tels que les cadres, les intervalles de temps et les contraintes de tranche permettant l'expression de la connaissance de manière orientée objet.

7. Par exemple : le terme "hypothèse" appartient à l'ontologie des tâches de diagnostic.

2.3.1.3 Déclaration - Triple

Les déclarations assignent des propriétés à des ressources. Une déclaration est un triple $ressource - attribut - valeur$ Les valeurs peuvent être des ressources ou des littéraux, les littéraux sont des valeurs atomiques (chaînes de caractères).

Un exemple de déclaration est :

« Jean est le propriétaire de Rex. »

Il est évident que cette déclaration peut être vue comme un triple $(x,\ P,\ y)$. En logique, $P(x,\ y)$ correspond donc à un prédicat binaire reliant l'objet x à l'objet y. En effet, RDF ne propose que des prédicats binaires appelées propriétés.

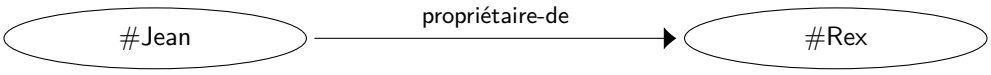

FIGURE 2.2 – Représentation graphique d'un triple

Dans cette exemple, la propriété relie deux ressources, cependant ce n'est pas toujours le cas. Une propriété peut également relier une ressource à un littéral. Dans RDF, l'utilisation de littéraux typés permet de fournir le type d'information.

Le triple *(#Jean, age, "53"^^http ://www.w3.org/2001/XMLSchema#integer)* permet de définir l'âge de *Jean* en utilisant un littéral typé. Cet exemple nous montre deux choses :

— l'utilisation de la notation ^^ pour indiquer le type d'un littéral.

— l'utilisation de types de données prédéfinis par XML Schema que nous aborderons dans la Section 2.4.

2.3.2 URI

Comme nous l'avons vu, chaque ressource a un URI [15], un identifiant de ressource uniforme ("Universal Resource Identifier"). Les URI peuvent être de deux types :

— URL (Uniform Resource Locator) qui identifie une ressource grâce à son chemin d'accès [16]. Par exemple : *http ://www.ietf.org/rfc/rfc2396.txt* identifie un document décrivant la syntaxe générique des URI.

— URN (Uniform Resource Name) est un URI qui identifie la ressource par un nom [16]. Par exemple : *urn :ISSN :0167-64230451450523* identifie le journal scientifique *Science of Computer Programming* par son numéro de série.

Cette idée d'utiliser les URI pour identifier les ressources et les propriétés est assez importante. Ce choix nous donne un schéma de nommage global et unique. L'utilisation d'un tel système réduit considérablement le problème des homonymes qui a toujours été un frein pour la représentation des données distribuées.

2.4 RDF Schema

RDF est un langage universel qui permet aux utilisateurs de décrire des ressources en utilisant leur propre vocabulaire. Cependant, il ne fait aucune hypothèse sur quelque domaine d'application que ce soit et ne définit donc pas la sémantique du dit domaine. C'est donc à l'utilisateur de définir cette sémantique par le biais de RDF Schema (RDFS).

2.4.1 Classe

Les classes RDFS sont des ensembles de ressources. Une classe regroupe toutes les instances de même type. Par exemple : la classe *Chien* regroupera tous les individus de type *Chien*, si *Rex* est un chien, il fera donc partie de la classe *Chien*.

RDFS décrit également la propriété de *rdfs :subClassOf* permettant de définir une hiérarchie au sein des classes.

Définition 2.1. (subClassOf). Soit deux classes A et B, B est sous-classe de A si et seulement si :

$$\forall x : x \in B \Rightarrow x \in A$$

Dans l'exemple ci-dessous, une sous-classe *chien* de la classe *animal* est définie.

```
1 <rdfs:Class rdf:ID="Animal" />
2
3 <rdfs:Class rdf:ID="Chien">
4   <rdfs:subClassOf rdf:resource="#Animal"/>
5 </rdfs:Class>
```

Listing 2.1 – Définition d'une sous-classe (RDF)

2.4.2 Propriété

Le concept de propriété RDFS est une relation entre ressources. RDFS permet, en plus de les définir, d'en contraindre leur domaine et leur range.

Le domaine d'une propriété est définie grâce à *rdfs :domain* qui est une instance de *rdf :Property* indiquant, pour une propriété, une classe dont les éléments sujets de cette propriété doivent être des instances. Par exemple : le triple

$$P \ \textit{rdfs :domain} \ C$$

nous informe que P est une instance de *rdf :Property*, C est une instance de *rdf :Class* et que les ressources étant les sujets d'un triple dont le prédicat est P sont des instances de C.

Similairement au domaine, le range d'une propriété est définie grâce à *rdfs :range* qui est une instance de *rdf :Property* indiquant, pour une propriété, une classe dont

les éléments objets de cette propriété doivent être des instances. Par exemple : le triple

$$P \ rdfs:range \ C$$

nous informe que P est une instance de *rdf :Property*, C est une instance de *rdf :Class* et que les ressources étant les objets d'un triple dont le prédicat est P sont des instances de C.

RDFS décrit également la propriété de *rdfs :subPropertyOf* permettant de définir, comme *rdfs :subClassOf* le fait pour les classes, une hiérarchie pour les propriétés. Par exemple : le triple

$$P \ rdfs:subPropertyOf \ Q$$

nous informe que P et Q sont des instances de *rdf :Property* et que P est une sous-propriété de Q.

Dans l'exemple suivant, est définie la classe *Personne* dont *Homme* est une sous-classe. Ensuite la propriété *parentDe* ainsi que sa sous-propriété *pèreDe* sont définies. Il ne reste plus qu'à indiquer que le domaine de *pèreDe* sont les instances de *Homme* et que le range sont les instances de *Personne*.

2.4.3 Types

```
1  <rdfs:Class rdf:ID="Personne" />
2
3  <rdfs:Class rdf:ID="Homme">
4    <rdfs:subClassOf rdf:resource="#Personne"/>
5  </rdfs:Class>
6
7  <rdf:Property rdf:ID="parentDe"/>
8
9  <rdf:Property rdf:ID="pèreDe">
10   <rdfs:subPropertyOf rdf:resource="#parentDe"/>
11   <rdfs:domain rdf:resource="#Homme"/>
12   <rdfs:range rdf:resource="#Personne"/>
13 </rdf:Property>
```

Listing 2.2 – Définition d'une sous-propiété, de son domaine et de son range (RDF)

2.5 OWL

Le W3C, dans le but de fournir un standard de représentation des ontologies, propose le langage OWL [5] [8]. OWL est un langage du Web sémantique basé sur

8. Web Ontology Language

la logique computationnelle afin de permettre aux connaissances exprimées via ce dernier d'être exploitées par des programmes informatiques. Les ontologies décrites grâce à ce standard peuvent être partagées et publiées vie le World Wide Web [2], ces ontologies peuvent également être référencées les unes par rapport aux autres.

Pour rappel, OWL fait partie de la troisième couche du Web sémantique (voir Figure 2.1), en effet, il est construit comme une extension de RDF [3] tout en disposant d'une syntaxe XML. Il est dérivé du langage d'ontologie Web DAML+OIL [17] également développé par le W3C.

2.5.1 Les sous-langages de OWL

Le langage OWL fournit trois sous-langages de niveau d'expressivité, de complétude et de complexité différents. Cette sous-division fournit aux différents types d'utilisateurs des standards spécifiques à leurs besoins respectifs.

2.5.1.1 OWL-Lite

OWL-Lite est adéquat pour les utilisateurs de base nécessitant une hiérarchie de classification et des contraintes simples [9]. Il est plus simple et rapide de fournir des outils pour OWL-Lite plutôt que pour ses parentés plus expressifs.

2.5.1.2 OWL-DL

OWL-DL fournit aux utilisateurs une expressivité maximale et la décidabilité [10] des systèmes de raisonnement. OWL-DL inclut toutes les constructions de langage OWL et est ainsi nommé en raison de sa correspondance avec les logiques de description (Description Logiques). OWL-DL a été conçu pour prendre en charge les activités de logique de description existantes et possède des propriétés de calcul nécessaires aux systèmes de raisonnement.

2.5.1.3 OWL-Full

OWL-Full a été conçu pour les utilisateurs souhaitant une expressivité maximale et la liberté syntaxique de RDF sans garantie de calcul. Contrairement à OWL-DL, il est peu probable qu'un logiciel de raisonnement puisse supporter toutes les fonctionnalités de OWL-Full.

2.5.1.4 Tableau de comparaison des sous-langages de OWL

En 2007, Ritesh Agrawal publiait une comparaison détaillée de OWL-Lite, OWL-DL et WOL Full dans [18]. Cet article peut être résumé en le tableau récapitulatif ci-dessous (Tableau 2.1).

9. Par exemple : OWL-Lite supporte les contraintes de cardinalité, cependant elles sont restreintes aux valeurs 0 et 1.

10. Tous les calculs se finissent en un temps imparti.

	OWL-Lite	**OWL-DL**	**OWL-Full**
Compatibilité avec RDF	Aucun document RDF n'est compatible.	Aucun document RDF n'est compatible.	Tout document RDF valide sont OWL-Full.
Restrictions sur la définition des classe	Nécessite la séparation des classes, des instances, des propriétés et des valeurs.	Nécessite la séparation des classes, des instances, des propriétés et des valeurs.	Les classes peuvent être des instances ou des propriétés en même temps [11].
Mélange RDF	Restreint les mélanges de constructions de RDF et de OWL.	Restreint les mélanges de constructions de RDF et de OWL.	Supporte et permet les mélanges de constructions de RDF et de OWL.
Description des classes	Supporte uniquement la description de classe : *IntersectionOf*	Supporte les descriptions de classe suivantes : *UnionOf*, *ComplementOf*, *IntersectionOf* et *Enumeration*.	Supporte les descriptions de classe suivantes : *UnionOf*, *ComplementOf*, *IntersectionOf* et *Enumeration*.
Contraintes de cardinalité	Cardinalité de 0 ou 1.	Cardinalité égale ou supérieure à 0.	Cardinalité égale ou supérieure à 0.
Méta-modelage	Ne permet pas le méta-modelage.	Ne permet pas le méta-modelage.	Permet le méta-modelage.
Classes	Les classes sont de types *OWL :class* et sont sous-type de *rdfs :class*.	Les classes sont de types *OWL :class* et sont sous-type de *rdfs :class*	Les classes sont de types *rdfs :class*.

TABLE 2.1 – Comparaison de OWL-Lite, OWL-DL et OWL-Lite

2.5.2 Les éléments de base

Une ontologie OWL comporte une multitude d'éléments dont la plupart concernent les classes, propriétés, instances de classe et relation entre ces instances. Ces éléments ont déjà été introduits dans la section 2.2 propre aux ontologies. Ces-derniers sont présentés grâce à leurs composants linguistiques respectifs.

2.5.2.1 Classes et individus

Les classes correspondent au concept de base du domaine, elles en représentent donc les racines des arbres. Il est important de préciser que dans OWL, tout individu est membre de la classe *owl :Thing*. Cela implique que toute classe créée par l'utilisateur est implicitement une sous-class de *owl :Thing*. Par défaut, OWL génère aussi la classe vide *owl :Nothing*.

```
<owl:Class rdf:ID="Mammifere"/>
<owl:Class rdf:ID="Homme">
  <rdfs:subClassOf rdf:resource="#Mammifere"/>
</owl:Class>
<owl:Class rdf:ID="Chien">
  <rdfs:subClassOf rdf:resource="#Mammifere"/>
</owl:Class>
```

Listing 2.3 – Création de classes et sous-classes (OWL)

Les membres des classes sont appelés les individus. Un individu est créé au minimum en le déclarant membre d'une classe.

```
<Homme rdf:ID="Marie"/>
<Chien rdf:ID="Rex"/>
```

Listing 2.4 – Création d'individus (OWL)

2.5.2.2 Propriétés

Les propriétés sont le moyen d'enrichir les classes et leurs individus en permettant d'introduire des faits généraux à propose des membres d'une classe et des faits spécifiques à propos des individus. Il existe deux types de propriétés, les propriétés d'objet et de donnée.

Les propriétés de donnée sont des relations entre des instances de classes et des littéraux RDF [12].

```
<owl:DatatypeProperty rdf:ID="age">
  <rdfs:domain rdf:resource="#Mammifere" />
  <rdfs:range rdf:resource="&xsd;positiveInteger"/>
</owl:DatatypeProperty>
```

Listing 2.5 – Description d'une propriété objet (OWL)

Les propriétés d'objet sont, quant à elles, des relations entre deux instances de deux classes.

```
<owl:ObjectProperty rdf:ID="possede">
  <rdfs:domain rdf:resource="#Humain"/>
  <rdfs:range rdf:resource="#Chien"/>
</owl:ObjectProperty>
```

Listing 2.6 – Description d'une propriété de donnée (OWL)

12. Un littéral est utilisé pour représenter des valeurs telles que des chaînes de caractères, des nombres ou encore des dates.

Finalement, après avoir décrit les différentes propriétés d'objet ou de donnée propres à certaines classes, il est possible de les définir pour des individus.

```
<Homme rdf:ID="Jean">
  <possede rdf:resource="#Rex"/>
  <age rdf:datatype="&xsd;positiveInteger">43</age>
</Homme>
```

Listing 2.7 – Définition d'un individu et de ses propriétés (OWL)

2.5.2.3 Caractéristiques de propriété

Après avoir créé des propriétés, il est possible de décrire des mécanismes pour spécifier d'avantage ces propriétés. Ces caractéristiques dès lors définies permettront le raisonnement certain à propos des propriétés. Les différentes caractéristiques relatives aux propriétés ainsi que leur définition sont énoncées ci-dessous.

Définition 2.2. (TransitiveProperty). Une propriété, P, est transitive si et seulement si :

$$\forall x,\ y\ et\ z : P(x,\ y) \wedge P(y,\ z) \Rightarrow P(x,\ z)$$

Définition 2.3. (IrreflexiveProperty). Si une propriété, P, est irréflexive alors :

$$\forall x\ \neg P(x,\ x)$$

Définition 2.4. (FunctionalProperty). Une propriété, P, est fonctionnelle si et seulement si :

$$\forall x,\ y\ et\ z : P(x,\ y) \wedge P(x,\ z) \Rightarrow y = z$$

Définition 2.5. (InverseOf). Soit deux propriétés, P et Q, si P est l'inverse de Q alors :

$$\forall x\ et\ y : P(x,\ y)\ si\ et\ seulement\ si\ Q(y,\ x)$$

Définition 2.6. (InverseFunctionalProperty). Une propriété, P, est inversement fonctionnelle si et seulement si :

$$\forall x,\ y\ et\ z : P(y,\ x) \wedge P(z,\ x) \Rightarrow y = z$$

Par exemple : si nous souhaitions créer une nouvelle propriété objet *possede* et la caractériser par le fait qu'elle est l'inverse de la propriété *nePossedePas* et qu'elle est inversement fonctionnelle, alors, le code suivant serait créé.

```
<owl:ObjectProperty rdf:ID="possede">
  <rdf:type rdf:resource="&owl;InverseFunctionalProperty" />
  <owl:inverseOf rdf:resource="#nePossedePas" />
```

```
4 </owl:ObjectProperty>
```

Listing 2.8 – Description d'une propriété objet et de ses caractéristiques (OWL)

2.5.2.4 Restrictions de propriété

En plus de définir des caractéristiques propres aux propriétés, OWL permet également de contraindre la portée de ces propriétés en utilisant des restrictions de propriété.

La principale restriction de propriété est la contrainte de cardinalité. Cette dernière permet, comme son nom l'indique, de définir la cardinalité d'une propriété.

Chapitre 3

Logiques descriptives et raisonnement

Au cœur de ce travail réside le raisonnement ontologique, ce dernier permet d'inférer des connaissances implicites à partir de connaissances explicites. Cependant, le raisonnement a besoin d'une description détaillée du domaine pour pouvoir être exécuté. Cette description est qualifiée de description logique et permet de définir des relations logiques entre différents concepts et individus.

Dans cette section, est abordée en premier lieu la description logique, nous abordons ensuite le langage de description qu'est le *Semantic Web Rule Language* (SWRL)[19]. Nous abordons enfin les deux types de raisonnement que sont les raisonnements *forward chaining* et *backward chaining* mais également leur fusion qualifiée de raisonnement hybride.

3.1 Description logique

Dans l'article [20], les logiques de description (DL) également appelées logiques descriptives sont définies comme une famille de langages de représentation de connaissance. Leur puissance expressive a pour but de fournir des primitives de modélisation de connaissances utiles tout en permettant des procédures de décision efficaces pour les problèmes de raisonnement. La logique de description a été conçue afin d'étendre certains langages ne possédant pas de sémantique formelle basée sur la logique. La logique de description est utilisée dans de nombreux domaines dont évidemment le Web sémantique et plus précisément la représentation d'ontologies.

Toute logique de description est répartie selon deux niveaux : le niveau des informations terminologiques et le niveau des informations sur les individus. Ces deux niveaux sont respectivement appelés T-Box et A-Box. La T-Box (*Terminological Box*) contient la définition d'un vocabulaire. La A-Box (*Assertional Box*) quant à elle contient les assertions, exprimées à partir du vocabulaire défini dans la TBox, relatives aux individus du domaine.

Les notions de concept, de rôle et d'individu sont utilisées par les logiques de description. Les concepts représentent une classe d'individus et sont interprétés comme des ensembles. Les rôles correspondent aux liens entre les éléments et peuvent être interprétés comme des relations binaires. Pour finir, les individus sont les éléments du domaine étudié. Il est dès lors possible de définir une signature et une interprétation de signature.

Définition 3.1. (Signature). Soit $CON = \{C_1, C_2, ...\}$ un ensemble fini de concepts atomiques, $ROL = \{R_1, R_2, ...\}$ un ensemble fini de rôles atomiques et $IND = \{a_1, a_2, ...\}$ un ensemble fini d'individus. Le triple $\langle CON,\ ROL,\ IND \rangle$ est appelé une signature et est noté S si et seulement si les ensembles CON, ROL et IND sont disjoints.

Définition 3.2. (Interprétation). Une interprétation I pour la signature S est un couple $I = \langle \Delta^I,\ .^I \rangle$ où :

- Δ^I est un ensemble non vide appelé domaine d'interprétation
- $.^I$ est une fonction qui :
  - associe un élément $a_i^I \in \Delta^I$ à chaque individu $a_i \in IND$
  - associe un sous-ensemble $C_i^I \subseteq \Delta^I$ à chaque concept atomique $C_i \in CON$
  - associe une relation $R_i^I \subseteq \Delta^I \times \Delta^I$ à chaque rôle atomique $R_i \in ROL$

Les définitions de signature et d'interprétation assimilées, nous pouvons définir précisément les T-Box et A-Box ainsi que leurs contenus respectifs.

Définition 3.3. (T-Box). La T-Box est un ensemble fini, potentiellement vide, d'expressions appelées axiomes terminologiques. Ces axiomes terminologiques sont de la forme :

$$C_1 \sqsubseteq C_2 \ ou\ C_1 \equiv C_2$$

où C_1 et C_2 sont des concepts.

Définition 3.4. (A-Box). La A-Box est un ensemble fini, potentiellement vide, de formules appelées assertions. Ces assertions sont de la forme :

$$a_1 : C_1 \ ou\ (a_1, a_2) : R_1$$

où a_1 et a_2 sont des individus, C_1 est un concept et R_1 un rôle.

Il existe de multiples logiques de descriptions, elles ont toute une base commune qu'elles enrichissent par différentes extensions telles que des concepts complexes composés de concepts atomiques ou rôles complexes composés de rôles atomiques.

Un langage de description logique peut être considéré comme la logique minimale de base pour les autres langages de description logique. Il s'agit du langage $\mathcal{AL}$ (*Attributive Language*) qui inclut dans sa fonction d'interprétation :

— le concept atomique universel : A

— le concept universel : $\top$ interprété par $\top^I = \Delta^I$

— le concept bottom : $\bot$ interprété par $\bot^I = \emptyset$

— la négation atomique : $\neg A$ interprétée par $\neg A^I = \Delta^I / A^I$

— l'intersection de concept : $C_1 \cap C_2$ interprétée par $(C_1 \cap C_2)^I = C_1^I \cap C_2^I$

— la restriction existentielle : $\exists R.C$

— la restriction universelle : $\forall R.C$

De multiples extensions existent pour enrichir les différentes logiques de description, le tableau 3.1 ci-dessous les décrit exhaustivement. Il est nécessaire de noté que ces dénominations sont propres à une convention de nommage informelle.

Extension	**Description**
$\mathcal{F}$	Propriété fonctionnelle.
$\mathcal{E}$	Quantificateur existentielle complet.
$\mathcal{U}$	Disjonction.
$\mathcal{C}$	Concept de négation complexe.
$\mathcal{H}$	Hiérarchie des rôles.
$\mathcal{R}$	Conjonction des rôles.
$\mathcal{O}$	Nomination, restriction de la valeur de l'objet (un-de).
$\mathcal{I}$	Propriété inverse.
$\mathcal{N}$	Restriction de cardinalité.
$\mathcal{Q}$	Restriction de cardinalité qualifiée.
$(\mathcal{D})$	Utilisation de propriétés de type, de valeur et de type de donnée.

TABLE 3.1 – Extensions pour les logiques de description

Dans le cadre ontologique de ce mémoire, il est important de préciser que OWL-DL abordé dans la section 2.5.1.2 est basé sur le langage de description logique $\mathcal{SHOIN}^{(D)}$ alors que OWL-Lite est, quant-à lui, basé sur le langage de description logique $\mathcal{SHIF}^{(D)}$. Grâce au tableau 3.1 ci-dessus, il est désormais aisé de comprendre et d'identifier les différentes extensions qui sont à la base de OWL-DL et OWL-Lite.

Nous pouvons désormais définir la notation conventionnelle, celle-ci est décrite de façon exhaustive dans le tableau 3.2 suivant.

Symbole	Description	Exemple
$\top$	Concept spécial dont tous les individus sont une instance.	$\top$
$\bot$	Concept vide.	$\bot$
$\sqcap$	Conjonction de concept.	$C_1 \sqcap C_2$
$\sqcup$	Disjonction de concept.	$C_1 \sqcup C_2$
$\neg$	Négation ou complément de concept.	$\neg C_1$
$\forall$	Quantificateur universelle.	$\forall R_1.C_1$
$\exists$	Quantificateur existentielle.	$\exists R_1.C_1$
$\sqsubseteq$	Inclusion de concept.	$C_1 \sqsubseteq C_2$
$\equiv$	Équivalence de concept.	$C_1 \equiv C_2$
$\doteq$	Définition de concept.	$C_1 \doteq C_2$
:	Assertion de concept.	$a_1 : C_1$
:	Assertion de rôle.	$(a_1, a_2) : R_1$

TABLE 3.2 – Notation conventionnelle des logiques de description

Nous pouvons, à partir des descriptions de $\mathcal{AL}$, des différentes extensions et de la notation conventionnelle, exposer quelques exemples :

— $\neg Animal$ décrit les individus qui ne sont pas des animaux.

— $Animal \cap Chien$ décrit les animaux qui sont des chiens.

— $Homme \cap \exists parentDe.\top$ décrit les Hommes qui sont au moins parents une fois.

3.2 SWRL

SWRL (Semantic Web Rule Language)[19] est un langage basé sur la combinaison de OWL-DL et OWL-Lite, il étend l'ensemble des axiomes de OWL avec des règles sous forme de clauses de Horn. Nous allons donc dans un premier temps redéfinir ce que sont les clauses de Horn, nous abordons dans un second temps la sémantique propres aux règles SWRL.

3.2.1 Clause de Horn

En mathématique et en programmation logique, une clause de Horn est une formule logique d'un type particulier qui la munit de propriétés très utiles[21].

Une clause de Horn est donc une clause [1] comprenant au plus un littéral positif. Il existe donc trois types de clauses de Horn que sont :

— les clauses de Horn comprenant un littéral positif et aucun littéral négatif. Les clauses de Horn de ce type sont appelées clauses de Horn positives ou simplement des faits. Les faits sont des variables propositionnelles représentant des propositions élémentaires pouvant être vraies ou fausses. En

1. Une clause est un disjonction de littéraux.

voici un exemple :

$$p$$

— les clauses de Horn comprenant un littéral positif et au moins un littéral négatif. Les clauses de Horn de ce type sont appelées clauses de Horn strictes. En voici un exemple :

$$p \vee \neg q \vee \neg r \vee ... \vee \neg s$$

— les clauses de Horn comprenant uniquement des littéraux négatifs. Les clauses de Horn de ce type sont appelées clauses de Horn négatives. En voici un exemple :

$$\neg p \vee \neg q \vee \neg r \vee ... \vee \neg s$$

Le principal avantage des clauses de Horn et plus précisément des clauses de Horn strictes est que ces dernières peuvent s'écrire sous une forme implicative. La clause de Horn suivante :

$$\neg p \vee \neg q \vee \neg r \vee s$$

peut donc s'écrire sous la forme implicative :

$$p \wedge q \wedge r \Rightarrow s$$

Dans le monde non-propositionnelle, toutes les variables sont implicitement quantifiées de manière universelle. Nous avons donc par exemple que :

$$\neg Homme(X) \vee Mammifere(X)$$

est équivalent à :

$$\forall X : \neg Homme(X) \vee Mammifere(X)$$

qui en logique et sous une forme implicative correspond à :

$$\forall X : Homme(X) \Rightarrow Mammifere(X)$$

3.2.2 Introduction à SWRL

Comme l'a été annoncé en tête de section, SWRL a été proposé pour enrichir les langages de descriptions logiques déjà connus pour les ontologies avec des règles de type clauses de Horn.

Les règles SWRL ont la forme d'une implication entre un antécédent (le corps de la règle) et d'un conséquent (la tête de la règle). Cette formulation implicative a pour

signification que : si les conditions spécifiées dans l'antécédent sont valides, alors les conditions spécifiées dans le conséquent sont valides également.

Les corps et têtes de règles SWRL sont composées d'atomes, si l'antécédent est vide, alors, il est considéré comme vrai, cela signifie que le conséquent doit être valide pour toute interprétation possible. Si le conséquent est vide, alors il est considéré comme faux et donc l'antécédent ne doit pas être satisfait par aucune interprétation.

Les atomes composant les corps et têtes des règles SWRL peuvent être de plusieurs formes :

— $C(x)$

— $P(x, y)$

— $sameAs(x, y)$

— $differentFrom(x, y)$

C est une description OWL, P est une propriété OWL et x et y sont des variables, des individus ou des valeurs.

3.2.3 Syntaxe de SWRL

Il est existe plusieurs syntaxes pour définir des règles SWRL dont une syntaxe concrète pour XML et une autre pour RDF, cependant, dans ce travail et dans la plupart des cas, une forme relativement informelle est retenue car elle est beaucoup plus lisible pour l'être humain[19]. Dans cette syntaxe familière à l'homme, une règle a la forme suivante :

$$corps \Rightarrow tete$$

où le corps est une conjonction d'atomes alors que la tête n'en comporte qu'un seul. Les variables présentes dans les règles sont indiquées en utilisant la convention qui consiste à ajouter un point d'interrogation en tant que préfixe. Ci-dessous est écrite une règle SWRL définissant facilement le concept d'oncle.

```
1 estParentDe(?x,?y) ^ estFrereDe(?x,?z) => estOncleDe(?z,?y)
```

Listing 3.1 – Concept de *estOncleDe* (SWRL)

L'expressivité de SWRL est un atout majeur, en effet, dans OWL, le concept d'oncle ne pouvait être défini que de cette façon :

```
1 intersectionOf(SubClassOf(Homme), estFrereDe(Pere)).
```

Listing 3.2 – Concept de *estOncleDe* (OWL)

3.3 Raisonnement *forward chaining*

3.3.1 Qu'est-ce que le *forward chaining* ?

Lorsqu'il est question de moteur d'inférence, le *foward chaining*, également appelé *foward reasoning*, est l'une des deux principales méthodes de raisonnement. Cette méthode de raisonnement est une méthode *bottom-up* car elle se base sur les connaissances de base pour remonter à travers les règles pour inférer de nouvelles connaissances. La seconde méthode de raisonnement s'appelle le *backward chaining* et est abordée dans la section 3.4.

Un moteur d'inférence *foward chaining* se base sur les données qui lui sont mises à disposition. À partir de ces données, le moteur d'inférence va parcourir les règles à la recherche de celles dont l'antécédent est validé par les données qui sont en sa possession. Une fois ces règles identifiées, le moteur d'inférence est capable d'inférer le conséquent de la règle. Ce conséquent génère de nouvelles données valides qui seront ajoutées à la base de connaissances du moteur d'inférence. Ce dernier n'arrêtera son exécution que lorsque l'ensemble des assertions possibles auront été inférées. Le moteur s'arrêtera donc lorsqu'il ne sera plus possible de trouver de règles pertinentes pour les données traitées.

À titre d'exemple, imaginons les règles suivantes :

1. $aboie(?x) \wedge mangeDesCroquettes(?x) \Rightarrow chien(?x)$
2. $chien(?x) \Rightarrow noir(?x)$
3. $canari(?x) \Rightarrow jaune(?x)$

Nous sommes également en possession de certaines données que sont les suivantes :

a. $aboie(Rex)$: l'individu Rex aboie
b. $mangeDesCroquettes(Rex)$: l'individu Rex mange des croquettes

Nous pouvons dès lors commencer l'exécution du moteur d'inférence *forward chaining*. Celui-ci va, dans un premier temps, parcourir les règles afin de sélectionner celles pertinentes. La seule règle pertinente pour le moment est la règle 1. Le moteur va substituer la variables $?x$ dans le corps de la règle et va déduire : $chien(Rex)$, l'individu Rex est donc un chien. Deuxième occurrence, le moteur d'inférence va reparcourir les règles avec les données préalablement inférées, ce dernier va retenir la règle 2 et, par le même procédé que pour la règle 1, va être capable d'inférer une nouvelle donnée : $noir(Rex)$, l'individu Rex est noir. À la fin de l'exécution, les données inférées sont ajoutées aux données préalables à l'exécution, la base de connaissances est désormais la suivante :

a. $aboie(Rex)$
b. $mangeDesCroquettes(Rex)$
c. $chien(Rex)$

d. $noir(Rex)$

La figure 3.1 suivante permet de schématiser le fonctionnement du moteur d'inférence *forward chaining*. Comme nous l'avons décrit ci-dessus, le moteur d'inférence *forward chaining* est *bottom-up* ce qui signifie qu'il va partir du bas (des connaissances initiales : *Initial fact*) afin de parcourir les règles. Plus le nombre de règles parcourues est important, plus les connaissances s'étendent et le trapèze s'élargit. L'exécution s'arrête au sommet lorsque le moteur d'inférence a parcouru l'ensemble des règles pertinentes. Le moteur d'inférence part donc des données, parcourt les règles et en infère des conclusions.

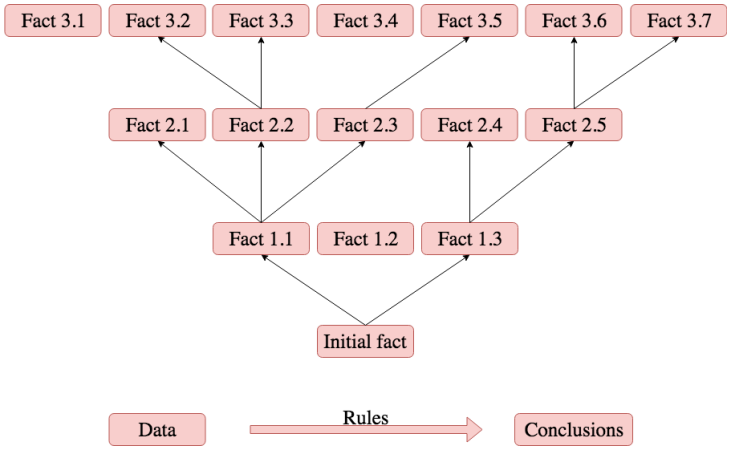

FIGURE 3.1 – Fonctionnement d'un moteur d'inférence *forward chaining*

La figure 3.2 suivante est utilisée dans le reste du document pour schématisé un moteur d'inférence *forward chaining* afin de simplifier la figure 3.1.

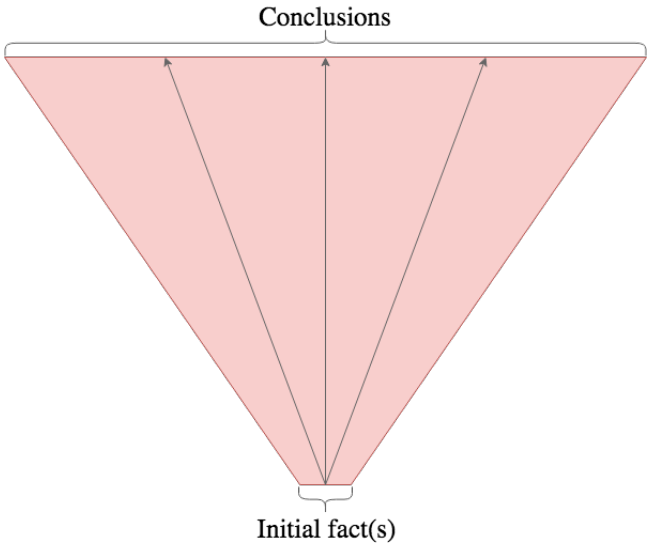

FIGURE 3.2 – Fonctionnement schématisé d'un moteur d'inférence *forward chaining*

3.3.2 Utilisation du *forward chaining* au sein du Web sémantique

Au sein de Web sémantique, il existe plusieurs moteurs d'inférence *forward chaining* tels que Pellet ou Bossam. Le framework Java Apache Jena en propose également un. Certains supportent SWRL alors que d'autres ont leur propre langage de règles.

3.3.2.1 Jena

Apache Jena [2] est un framework Java pour développer des applications pour le Web sémantique, il possède un sous-système d'inférence permettant l'utilisation de plusieurs moteurs d'inférence différents et notamment un moteur d'inférence *forward chaining* [22]. Ces moteurs sont utilisés pour dériver des assertions RDF [3] supplémentaires qui sont inférés à partir de certains RDF de base. L'utilisation principale de ce mécanisme est de soutenir l'utilisation de langages tels que RDFS et OWL qui permettent d'inférer des faits supplémentaires à partir de données d'instance et de descriptions de classes.

La structure générale du mécanisme d'inférence de Jena est décrite dans la figure 3.3 suivante :

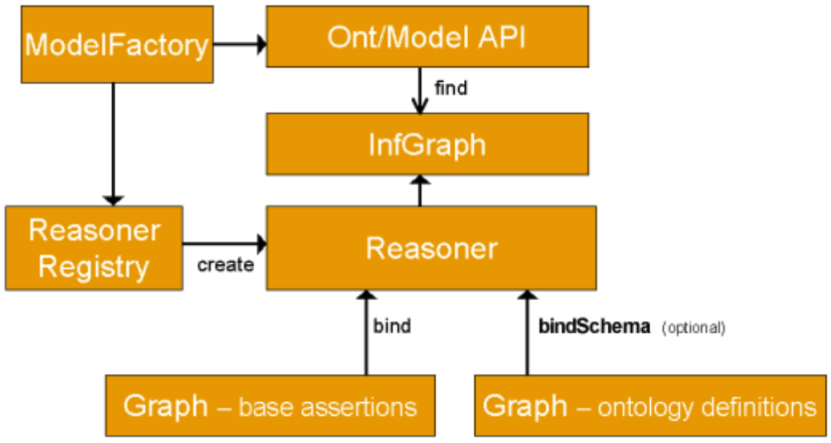

FIGURE 3.3 – Structure générale du mécanisme d'inférence de Jena
Source: [22]

La machinerie d'inférence est en fait implémentée de sorte que n'importe quelle interface du modèle puisse être construite autour d'un graphique d'inférence.

Lorsque le modèle d'inférence *forward chaining* est sélectionné, toutes les données pertinentes du modèle seront soumises à ce dernier. Toute règle qui déclenche des triples supplémentaires le fait dans un graphique de déductions internes et peut à son tour déclencher des règles supplémentaires.

2. Cette section ne décrit pas rigoureusement le fonctionnement du moteur d'inférence Jena, pour de plus amples informations, voir [22].
3. Ou triples RDF 2.3

Le graphique d'inférence agit comme s'il s'agissait de l'union de tous les énoncés du modèle original et de tous les énoncés du graphique des déductions internes générés par les règles. Il est possible d'accéder séparément aux données brutes initiales ainsi qu'à l'ensemble des déclarations déduites.

Au sein de Jena, il n'existe qu'une seule représentation pour les règles, quelque soit le type de moteur d'inférence utilisé, les règles et leur modèle sont identiques. De plus aucun ordre ne peut être garanti, ni quant au parcours des règles, ni quant à la résolution des éléments contenus dans les conjonctions présentes dans les antécédent des règles.

3.3.2.2 Pellet

Pellet [4] [23] est un moteur d'inférence *forward chaining* libre basé sur Java pour $\mathcal{SROIQ}$ avec des types de données simples. Il met en œuvre une procédure de décision basée sur des tableaux pour les T-Box générales (subsomption, satisfaisabilité et classification) et les A-Box (récupération, réponse à des requêtes conjonctives). Pellet utilise de nombreuses optimisations pour le raisonnement DL standard telles que la normalisation, la simplification et l'absorption. Il supporte directement les contrôles d'implication et l'interrogation optimisée d'A-Box à travers son interface.

La figure 3.4 représente la structure principale du moteur d'inférence Pellet, en son cœur se trouve un raisonneur basé sur les tableaux. Les *tableaux reasoners* ne sont pas abordés dans ce travail.

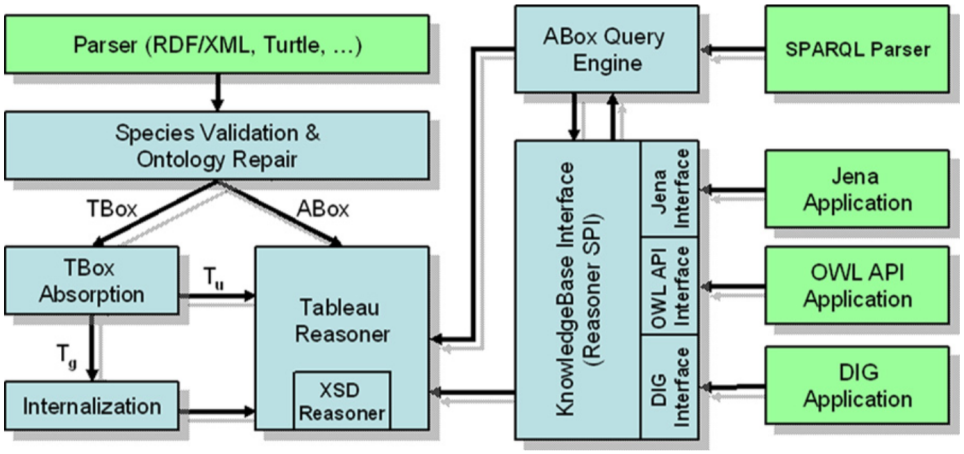

FIGURE 3.4 – Composants principaux du moteur d'inférence *forward chaining* Pellet
Source: [23], p.2

Pellet supporte les interfaces OWL-API, DIG-API et Jena.

4. Cette section ne décrit pas rigoureusement le fonctionnement du moteur d'inférence Pellet, pour de plus amples informations, voir [23].

3.3.2.3 Bossam

Bossam [5] [24] est un moteur d'inférence *forward chaining*, il est équipé de fonctions de représentation et de fonctions extra-logiques telles que :

— la prise en charge de la négation en tant qu'échec (*negation-as-failure*) et de la négation classique,

— la limitation de la portée dans les têtes des règles,

— la liaison à distance pour l'inférence coopérative entre plusieurs moteurs de règle.

La figure 3.5 illustre l'expressivité de Bossam. Le rectangle externe est la limite de l'expressivité de la logique du premier ordre. La logique de description et la logique de Horn forment deux fragments qui se chevauchent à l'intérieur de la logique de premier ordre. La programmation logique est principalement inclue dans la logique du premier order, mais elle contient des caractéristiques extra-logiques, telles qu'entre autres la *negation-as-failure* et la résolution de conflit.

L'expressivité de Bossam correspond aux éléments : (2) + (3) + (4) + (5) + (6).

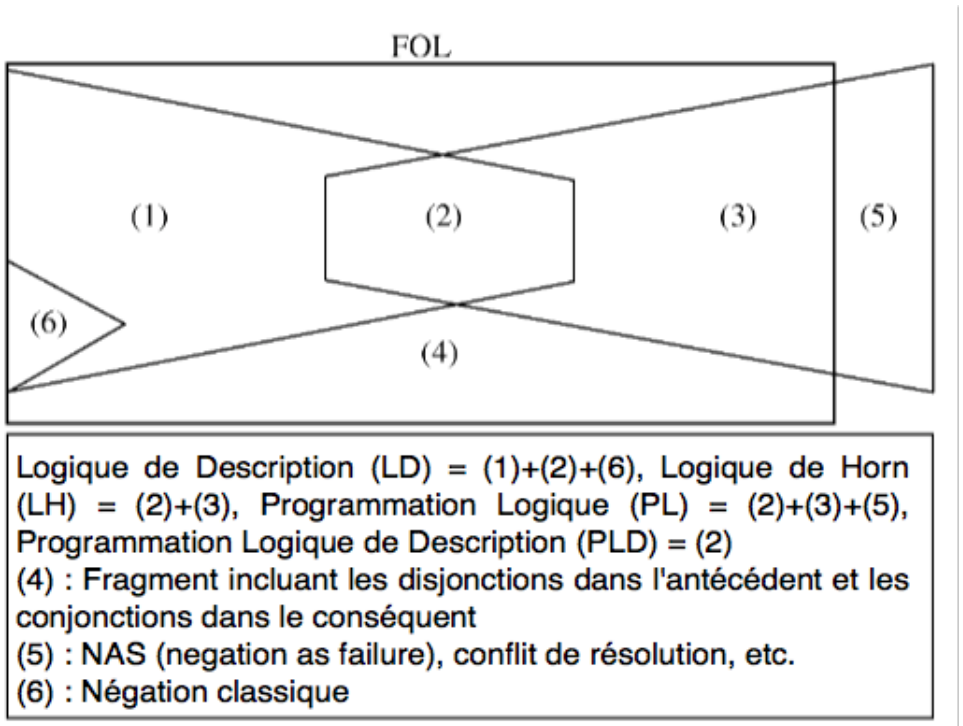

FIGURE 3.5 – Expressivité du moteur d'inférence *forward chaining* Bossam
Source: [24], p.130

Comme l'explique la figure 3.5, Bossam est basé sur la programmation logique, avec deux fragments d'expressivité ajoutés :

5. Cette section ne décrit pas rigoureusement le fonctionnement du moteur d'inférence Bossam, pour de plus amples informations, voir [24].

— (4) : fournit une commodité syntaxique utile pour la rédaction de règles concises. Bossam décompose les règles contenant les éléments du fragment (4) en plusieurs clauses de Horn.

— (6) : les moteurs d'inférence ordinaires sont basés sur l'hypothèse d'un monde fermé et ne peuvent pas représenter et traiter correctement la négation classique. Bossam, quant à lui, comprend deux symboles pour désigner les négations *not* pour la *negation-as-failure* et *neg* pour la négation classique.

3.4 Raisonnement *backward chaining*

3.4.1 Qu'est-ce que le *backward chaining* ?

La deuxième principale méthode de raisonnement, lorsqu'il est question de moteur d'inférence, est le *backward chaining*. Cette méthode de raisonnement est dite *top-down* car, à partir d'une requête [6], elle va parcourir les règles jusqu'à pouvoir prouver ou non la requête émise.

Un moteur d'inférence *backward chaining* a besoin d'une requête pour son exécution. À partir de cette requête, le moteur d'inférence va parcourir les règles à la recherche de celles dont le conséquent est pertinent [7] pour la requête. Une fois ces règles identifiées, le moteur d'inférence va parcourir l'antécédent afin de voir si les conditions spécifiées dans ce dernier sont valides. Si ces dernières sont valides alors le conséquent l'est aussi et la requête est valide. Si le moteur d'inférence n'est pas capable de définir si le conséquent est valide ou pas, une nouvelle itération sera exécutée afin de trouver d'autres règles qui pourraient valider la requête. L'exécution se termine lorsque le moteur d'inférence a pu identifier une arborescence à travers les règles permettant de relier la requête à des connaissances explicites, la requête a donc pu être résolue. Cependant l'exécution peut également se terminer en ne trouvant pas d'arborescence ou en recontrant une boucle infinie, dès lors la requête échoue.

À titre d'exemple, imaginons les règles suivantes [8] :

1. $aboie(?x) \wedge mangeDesCroquettes(?x) \Rightarrow chien(?x)$
2. $chien(?x) \Rightarrow noir(?x)$
3. $canari(?x) \Rightarrow jaune(?x)$

Nous sommes également en possession de certaines données que sont les suivantes :

a. $aboie(Rex)$
b. $mangeDesCroquettes(Rex)$

6. Une requête est une proposition à vérifier.
7. Un conséquent est pertinent s'il est composé d'un atome ayant le même prédicat que celui présent dans la requête.
8. Ces règles sont les mêmes que celles utilisées pour l'exemple du moteur d'inférence *forward chaining*.

Contrairement au moteur d'inférence *forward chaining*, nous ne pouvons pas commencer l'exécution sans but initial. Le moteur d'inférence *backward chaining* a besoin d'une requête pour commencer son exécution. Cette requête sera la suivante : $noir(Rex)$, il nous est donc demandé de vérifier si l'individu Rex est de couleur noire.

Celui-ci va, dans un premier temps, parcourir les règles afin de sélectionner celles pertinentes, c'est-à-dire les règles dont le prédicat *noir* apparaît dans le conséquent. La seule règle pertinente est la règle 2. Le moteur a désormais pour objectif de valider l'antécédent afin de pouvoir affirmer que le conséquent. Or il n'est pas en présence du fait que l'individu Rex soit un chien : $chien(Rex)$. Une nouvelle itération va donc être exécutée afin de trouver des règles qui pourraient inférer le fait : $chien(Rex)$, la règle 1 va être sélectionnée. Cette fois le moteur d'inférence *backward chaining* est capable de vérifier le conséquent étant donné qu'il a dans sa base de connaissances les informations suivante : $aboie(Rex)$ et $mangeDesCroquettes(Rex)$. Le moteur est donc capable d'affirmer que Rex est un chien grâce à la règle 1 et par conséquent d'affirmer que Rex est noir grâce à la règle 2. Une réponse positive peut donc être donnée à la requête, $noir(Rex)$ est donc bien valide. Après l'exécution, la base de connaissances du moteur d'inférence a été étendue avec les données inférées lors de l'exécution, elle est donc identique à celle du moteur d'inférence *foward chaining* que nous avons détaillé dans la section 3.3.1 en fin d'exécution :

a. $aboie(Rex)$
b. $mangeDesCroquettes(Rex)$
c. $chien(Rex)$
d. $noir(Rex)$

Dans le cadre de cet exemple trivial, les bases de connaissances finales des deux moteurs sont identiques, ce n'est évidemment pas toujours le cas, les bases de connaissances finales sont généralement différentes car les moteurs ne parcourent pas les mêmes règles.

La figure 3.6 suivante permet de schématiser le fonctionnement du moteur d'inférence *backward chaining*. Comme nous l'avons décrit ci-dessus, le moteur d'inférence *backward chaining* est *top-down* ce qui signifie qu'il va partir du haut (de la requête initiale : *Request(s)*[9]) afin de parcourir les règles de haut en bas en analysant les conséquents en premier lieu. Le but du moteur d'inférence *backward chaining* est de créer une arborescence à travers les règles afin d'atteindre des connaissances explicites qui lui permettront de valider la requête. Dans la figure 3.6, les connaissances explicites peuvent sont représentées par les faits intitulés : *Fact 3.x.* Dès que l'arborescence a été identifiée, tous les faits inclus dans celle-ci peuvent être validés. Dans certains cas, aucune arborescence menant aux faits explicites ne peut être identifiée, dès lors aucune assertion ne peut être inférée.

9. Il peut éventuellement avoir plusieurs requêtes initiales.

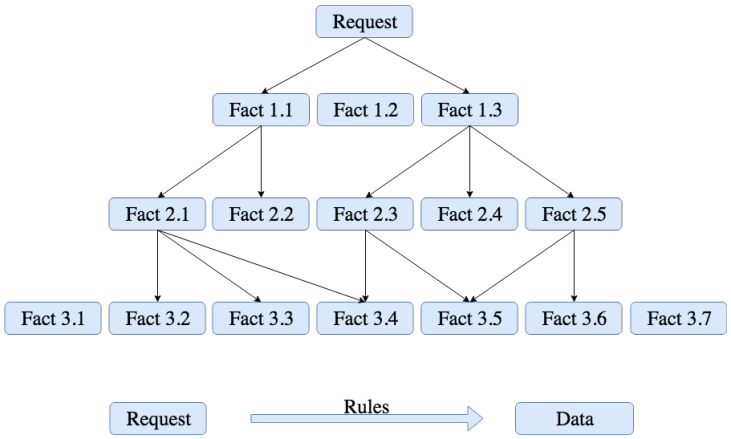

FIGURE 3.6 – Fonctionnement d'un moteur d'inférence *backward chaining*

La figure 3.7 suivante est utilisée dans le reste du document pour schématisé un moteur d'inférence *backward chaining* afin de simplifier la figure 3.6.

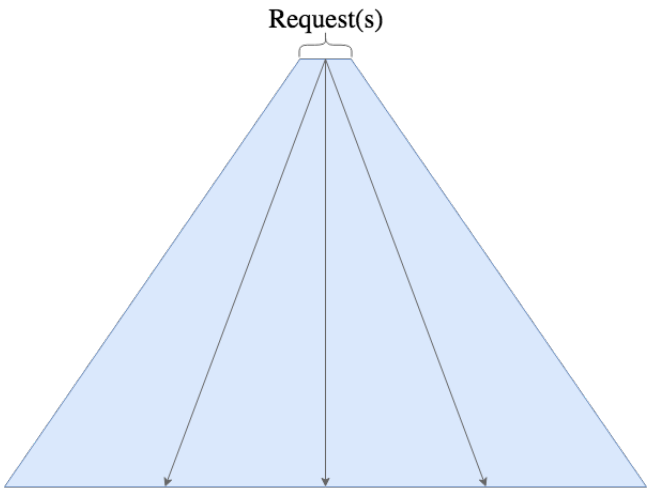

FIGURE 3.7 – Fonctionnement schématisé d'un moteur d'inférence *backward chaining*

3.4.2 Utilisation du *backward chaining* au sein du Web sémantique

Au sein du Web sémantique, les moteurs d'inférence sont souvent de type *forward chaining*. Cependant quelques moteurs *backward chaining* existent et Apache Jena en propose un.

3.4.2.1 Jena

Cette section reprend la description du moteur d'inférence *backward chaining* de Jena présente dans l'article [22]. Lorsqu'une requête est émise au moteur d'inférence,

elle est traduite en un but et le moteur tente de satisfaire ce but en faisant correspondre les triples stockés avec des règles en amont.

Les règles seront exécutées dans l'ordre de haut en bas et de gauche à droite. Le langage de règles est essentiellement du Datalog [10] plutôt que Prolog [11], alors que la syntaxe des foncteurs dans les règles permet la création de structures de données imbriquées, elles sont plates [12] et peuvent donc être considérées comme un sucre syntaxique pour Datalog.

En tant que langage d'enregistrement de données, la syntaxe des règles est un peu surprenante parce qu'elle restreint toutes les propriétés à être binaires [13] et permet aux variables d'occuper n'importe quelle position, y compris la position de la propriété. En effet, les règles de la forme :

$$(s,\ p,\ o),\ (s1,\ p1,\ o1),\ ...\ \Leftarrow (sb1,\ pb1,\ ob1)$$

peuvent être considérées comme étant traduites en règles d'enregistrement de données de la forme :

$$triple(s,\ p,\ o)\ :-\ triple(sb1,\ pb1,\ ob1),\ ...$$

$$triple(s1,\ p1,\ o1)\ :-\ triple(sb1,\ pb1,\ ob1),\ ...$$

$$...$$

où *triple* est un prédicat implicite caché. En interne, cette transformation n'est pas réellement utilisée.

En outre, toutes les données du modèle brut fourni au moteur sont traitées comme s'il s'agissait d'un ensemble de faits $triple(s, p, o)$ qui sont ajoutés à l'avant de l'ensemble des règles. Cependant, la mise en œuvre ne fonctionne pas de cette façon, mais consulte directement le graphique source, avec toutes ses capacités de stockage et d'indexation.

Parce que l'ordre des triples dans un modèle n'est pas défini, l'exécution se fait de haut en bas, tous les faits de base sont consultés avant toutes les clauses de la règle, mais l'ordre des faits de base est arbitraire.

10. Datalog est un langage de requête et de règles pour les bases de données déductives. Il correspond à un sous ensemble de Prolog.
11. Prolog est un langage de programmation logique
12. Plate signifie non-récursive.
13. De façon similaire à RDF

3.5 Raisonnement hybride

3.5.1 Qu'est-ce que le raisonnement hybride ?

La figure 3.8 ci-dessous représente schématiquement le fonctionnement d'un moteur d'inférence hybride. Cette figure correspond à une superposition des figures 3.2 et 3.7.

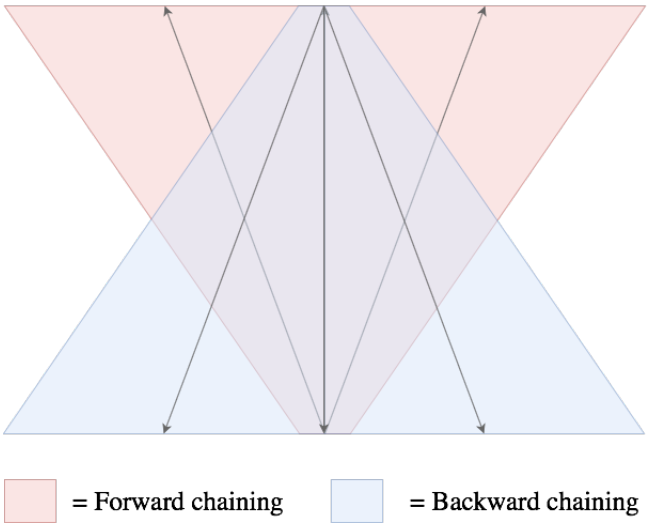

FIGURE 3.8 – Fonctionnement schématisé d'un moteur d'inférence hybride

L'idée derrière ce type de raisonnement est le gain de performance en combinant à la fois les moteurs d'inférence *forward chaining* avec les moteurs d'inférence *backward chaining*. L'origine de ce gain de performance sont la suppression d'inférences inutiles et le partage optimisé des règles entre les deux raisonneurs [14]. Ces inférences inutiles lors de la résolution d'une requête sont représentées dans la figure 3.8 par les quatre triangles externes.

En effet, les moteurs d'inférence pouvant être exécutés soit parallèlement, soit le moteur d'inférence *forward chaining* préalablement au moteur d'inférence *backward chaining*, permettent de supprimer ces inférences inutiles. Dans le cas où le moteur d'inférence *forward chaining* est exécuté préalablement au moteur d'inférence *backward chaining* alors, lorsque ce dernier recevra une requête, il sera capable de vérifier la véracité de celle-ci rapidement étant donné qu'une grande quantité d'assertions auront été inférées par le moteur d'inférence *forward chaining*.

Cependant la combinaison de ces deux types de raisonnements n'est pas si aisée qu'il n'y paraît, en effet de nombreuses questions se posent telles que :

— Quelle représentation de règle choisir ?

— Quelles connaissances initiales fournir au moteur *forward chaining* ?

14. Certaines règles sont parfois plus adaptées à un type de raisonnement qu'à l'autre.

— Quand exécuter les différents moteurs ?

— etc.

Ce travail a pour but d'essayer de répondre à ces questions via une approche expérimentale décrite dans la partie III.

3.5.2 Utilisation du raisonnement hybride au sein du Web sémantique

L'utilisation du raisonnement au coeur du Web sémantique est relativement restreinte, l'un des seuls moteurs d'inférence à utiliser ce type de raisonnement est, comme pour les moteurs d'inférence *backward chaining*, Jena.

3.5.2.1 Jena

Cette section reprend la description du moteur d'inférence hybride de Jena présente dans l'article [22].

Le raisonneur de règles de Jane offre la possibilité d'utiliser les deux moteurs de règles [15] individuels en conjonction. Lorsque cette option est sélecitonnée, les flux de données correspondent à la figure 3.9 ci-dessous.

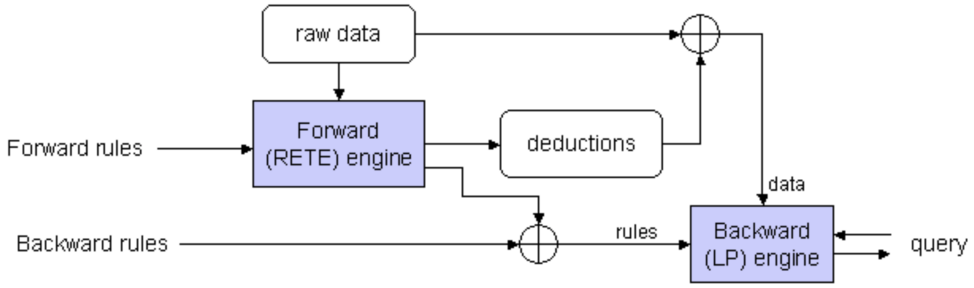

FIGURE 3.9 – Flux de données du raisonneur hybride Jena
Source: [22]

Le moteur d'inférence *forward* s'exécute de façon similaire à ce qui est décrit dans la section 3.3.2.1 et maintient un ensemble d'états déduits dans le magasin de déductions [16]. Toute règle *forward* qui affirme de nouvelles règles *backward* instanciera ces règles et les transmettra au moteur *backward*.

Les requêtes sont traitées en utilisant le moteur LP *backward chaining*, en utilisant les règles fournies et générées sur la fusion des données brutes et des données déduites préalablement par le moteur *forward*.

15. *Forward chaining* et *backward chaining*
16. Ce magasin de déductions peut être vu comme une cache.

Ce fractionnement permet au développeur de règles d'obtenir de meilleures performances en n'incluant que des règles *backward* pour l'ensemble de données en question. En particulier, Jena peut utiliser les règles *forward* pour compiler un ensemble de règles *backward* à partir des informations ontologiques. Comme exemple simple, essayez de mettre en œuvre la propriété *subPropertyOf* de RDFS à l'aide d'un moteur de règles. Une approche simple impliquerait des règles du type suivant :

$$(?a,\ ?q,\ ?b) \Leftarrow (?p,\ rdfs:subPropertyOf,\ ?q), (?a,\ ?p,\ ?b)$$

Une telle règle fonctionnerait mais chaque but correspondrait à la tête de cette règle et ainsi chaque requête invoquerait un test dynamique pour savoir s'il y avait une sous-propriété de la propriété recherchée. Au lieu de cela, la règle hybride :

$$(?p,\ rdfs:subPropertyOf,\ ?q),\ notEqual(?p,\ ?q) \Rightarrow [(?a,\ ?q,\ ?b) \Leftarrow (?a,\ ?p,\ ?b)]$$

pré-compile toutes les relations *subPropertyOf* déclarées dans des règles de chaîne simples qui ne se déclenchent que si le but de la requête fait référence à une propriété qui possède réellement une sous-propriété. S'il n'y a pas de relations *subPropertyOf*, il n'y aura donc pas de traitement pour une telle règle.

Notez qu'il n'y a pas de boucles dans les flux de données ci-dessus. Les règles *backward* ne sont pas utilisées lors de la recherche de correspondances avec des termes de règles *forward*. Cette exécution en deux phases est simple à comprendre et permet de garder la sémantique des moteurs de règles simples.

Troisième partie

Contribution personnelle

Chapitre 4

Problématique

Au sein du Web sémantique, comme nous l'avons décrit dans les sections 3.3, 3.4 et 3.5, de multiples moteurs d'inférence existent cependant, rares sont ceux utilisant le raisonnement hybride.

La problématique de ce travail consiste en l'implémentation et l'intégration dans raisonneur *forward* avec un raisonneur *backward*.

Nous aborderons dans le chapitre 5 suivant l'ensemble des enjeux et des questions primordiales posées pour cette exercice. Nous tenterons également d'y fournir des réponses de manière expérimentale.

Concernant ce chapitre sur la problématique, nous décrirons dans un premier temps le problème ainsi que son contexte et les concepts que nous avons à disposition. Dans un dernier temps sera abordée la méthodologie adoptée dans le cadre de cette recherche expérimental.

4.1 Description du problème

Ce travail a pour but d'explorer les éléments de réponse à la question suivante :

> *« Comment intégrer et combiner un raisonneur forward chaining avec un raisonneur backward chaining ? »*

Cette question générale en révèle bien d'autres beaucoup plus précises à propos, par exemple, de l'agencement des exécutions, de la structure des règles, etc. Ces questions spécifiques sont abordées en détail dans le chapitre 5.

4.2 Contexte

L'intégralité des recherches et des solutions sont codées en Mercury[25]. Mercury [1] est un langage de programmation logique et fonctionnel qui combine la clarté et l'expressivité de la programmation déclarative avec des fonctions avancées d'analyse statique et de détection d'erreurs.

Le contexte initial peut être schématisé par la 4.1 figure suivante :

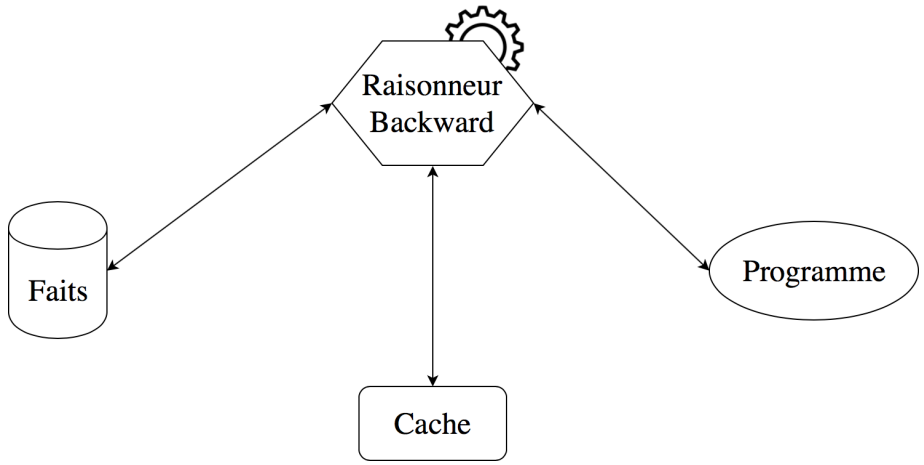

FIGURE 4.1 – Contexte initial

Comme vous pouvez le remarquer, les recherches ont débuté en se basant sur un système muni de quatre éléments principaux que sont :

— un raisonneur *backward chaining*
— une base de données comprenant les faits
— un programme correspondant à la représentation des règles
— une cache enregistrant temporairement les assertions déduites lors de la résolution d'une requête

Cette section a pour but de décrire précisément chaque entité présente dans la figure 4.1. Nous commencerons par décrire les éléments externes pour terminer par le raisonneur *backward* central.

4.2.1 Les faits

Les faits sont accessibles par le raisonneur via une interface qu'il n'est pas nécessaire de décrire dans le cadre de ce travail. Les faits ainsi que leur structure au sein de la base de données ne sont pas des éléments essentiels ni primordiaux à la réalisation de cette exercice.

1. Un très bon livre tutoriel pour Mercury est en cours de rédaction, ce dernier est disponible à l'adresse suivante : `https://www.mercurylang.org/documentation/papers/book.pdf`

Cependant, les faits, importés au cœur du raisonneur, sont importants et doivent être décrits. Ces derniers ont la forme de triple RDF décrits dans la section 2.3. Au sein du programme Mercury, voici la définition du type d'un triple RDF.

```
1  :- type triple(T)
2      ---> triple(
3              subject :: subject,
4              predicate :: predicate,
5              object :: object(T)).
6
7  :- type subject == resource.
8
9  :- type predicate ---> uri(p_value :: uri.uri).
10
11 :- type object(T)
12     ---> resource(obj_resource :: resource)
13         ;literal(obj_literal :: T).
14
15 :- type resource
16     ---> uri(uri.uri)
17         ;anon(rdf.value).
18
19 :- type literal
20     ---> literal(
21             l_value :: string,
22             l_type :: literal_type
23         ).
```

Listing 4.1 – Type d'un triple RDF (Mercury)

Vous pouvez le remarquer, un triple est composé de trois éléments : un sujet, un prédicat [2] et un objet.

Le sujet est une ressource qui elle-même est soit un URI [3] soit une valeur.

Le deuxième membre du triple qu'est le prédicat est lui aussi de type URI.

L'objet, quant-à lui peut, de façon similaire au sujet, être une ressource ou un littéral [4]. Ce littéral est composé d'une valeur sous la forme d'une chaîne de caractères et d'un type de littéral.

L'ensemble des faits sont regroupés selon des *stores*, ceux-ci sont munis d'identifiants afin d'y accéder aisément. L'explication et la compréhension de la structure de ces *stores* n'est pas nécessaire pour ce mémoire.

2. Un prédicat est une fonction mettant des arguments en relation et dont le résultat est vrai ou faux.
3. Cf. 2.3.2
4. Un littéral correspond à une valeur donnée explicitement

4.2.2 Le programme

Le programme est la représentation des règles. Dans le contexte actuel d'un raisonneur *backward chaining* les règles ont la structure suivante :

```
1  :- type rule(P,U,V)
2      ---> rule(
3              head :: rule_head(P,U,V),
4              body :: list(literal(P,U,V)),
5              name :: rule_name
6              ).
7
8  :- type rule_head(P, U, V)
9      ---> atom(atom(P, U, V)).
10
11 :- type atom(P,U,V)
12     ---> atom(
13             predicate :: P,
14             arguments :: list(term(V,U))
15             ).
16
17 :- type term(V,T)
18     ---> term_var(V)
19         ;term_value(T).
```

Listing 4.2 – Représentation des règles SWRL au sein du programme (Mercury)

Une règle est composée de trois éléments :

— une tête également appelé le conséquent,

— un corps également appelé l'antécédent,

— et un nom.

Une tête de règle est composée d'un seul et unique atome, cet atome est composé de deux éléments que sont le prédicat et une liste de termes. Cette liste de termes correspond aux arguments présents dans le prédicat. Ces derniers peuvent soit être une variable (*term_var*) soit une valeur (*term_value*).

Les règles sont indexées dans le but d'optimiser les performances du raisonneur *backward chaining*. Cette indexation est une multimap [5] associant un prédicat aux règles qui le contiennent dans leur conséquent.

4.2.3 La cache

La cache utilisée par le raisonneur *backward chaining* a pour but d'enregistrer temporairement les assertions déduites par le raisonneur. Ces assertions correspondent

5. Une multimap est la généralisation du type de données abstraites map dans lequel plus d'une valeur peut être associée à un clé.

à un couple associant à chaque prédicat des substitutions. Les substitutions associées à un prédicat contiennent les sujet et objet de ce prédicat. Par exemple, le prédicat $estPereDe$ peut être associé à plusieurs substitutions contenant dans l'ordre le sujet et l'objet. Le prédicat $estPereDe$ peut donc être associer à une liste de substitutions de ce type :

$$[(pere1,\ enfant1),\ (pere2,\ enfant2)]$$

Cependant ces informations ne suffisent pas, en effet, lors de son exécution, le raisonneur *backward* aura recourt à cette cache afin d'obtenir les différents individus répondant à un prédicat précis. Une information supplémentaire lui est nécessaire, celle de savoir si les informations sont complètes ou non. Le raisonneur *backward* veut pouvoir savoir si la liste de substitutions associée au prédicat qui l'intéresse est complète ou risque d'être modifiée. Le listing 4.4 définit le type pour les valeurs de retour associées à un prédicat.

```
1 :- type cache_value
2     ---> complete_value(set(canon_subst))
3     ; incomplete_value(incomplete_answers(canon_subst)).
```

Listing 4.3 – Type pour valeur présente dans la cache (Mercury)

Comme le montre le listing 4.4, les prédicats peuvent être associés à deux types de valeur que sont :

— la valeur complète comprenant un ensemble de substitutions. Cette valeur est dite complète car toutes les assertions propres au prédicat associé à cette valeur sont connues et présentes dans cet ensemble de substitutions.

— la valeur incomplète comprenant elle aussi un ensemble [6] de substitutions. Cette valeur, contrairement à la valeur complète, n'affirme pas que l'ensemble des assertions a été déduit. Le raisonneur, lorsqu'un tel résultat lui est fourni, sait qu'il doit potentiellement continuer son exécution afin d'obtenir un résultat complet.

Les substitutions (*canon_subst*) sont quant-à elles des listes d'objets RDF étant eux-mêmes des objets de type littéraux. Ces objets peuvent donc être, comme le définit le listing 4.1 soit une ressource soit un littéral.

```
1 :- type canon_subst
2       ---> canon_subst(
3               cs_subst :: list(rdf.object)
4           ).
5
6 :- type object == object(rdf.literal).
```

6. Cet ensemble est inclus dans le type *incomplete_answers(canon_subst)*

Listing 4.4 – Type d'une substitution (Mercury)

La représentation de cette cache et son implémentation incluent les principes de la *tabled execution* également appelée résolution SLG. L'article [26] décrit un moteur asbtrait supportant le *tabling* et plus précisément les résolutions SLG.

4.2.4 Le raisonneur *backward*

Cette section consiste en la description du raisonneur *backward chaining* en lui même. Il s'agit d'un moteur de règle SWRL[7] *depth-first*[8]. *Depth-first* signifie que le raisonneur va, lors de son exécution, parcourir les règles en profondeur d'abord. Lorsqu'il va trouver une règle cohérente pour la résolution de sa requête, ce dernier va directement chercher les règles cohérentes pour résoudre l'antécédent de la règle qu'il vient de trouver. Si le raisonneur n'était pas *depth-first*, il chercherait l'ensemble des règles cohérentes à sa requête avant d'en résoudre les antécédents. La figure 4.2 compare les deux types de recherches, la recherche *depth-first* (de haut en bas) et la recherhe *breadth-first*[9] (de gauche à droite).

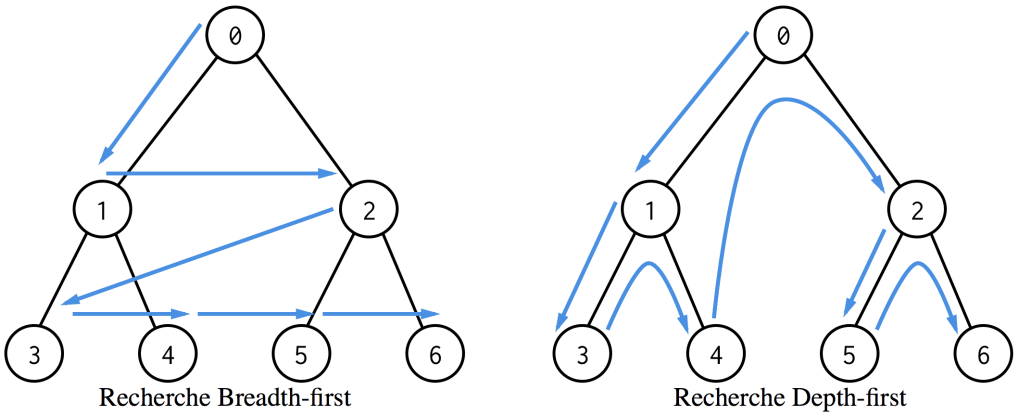

FIGURE 4.2 – Comparaison des recherches depth-first et breadth-first
Source: http ://mishadoff.com/blog/dfs-on-binary-tree-array/

Le raisonneur reçoit une requête sous la forme d'un atome, les atomes ont la structure suivante :

```
1 :- type atom(P,U,V)
2     ---> atom(
3         predicate :: P,
```

7. Voir la section 3.2 pour plus d'informations concernant SWRL.
8. *Depth-first*, en français : "en profondeur d'abord".
9. *Breadth-first* en français : "en largeur d'abord".

```
4        arguments :: list(term(V,U))
5    ).
```

Listing 4.5 – Type d'un atome (Mercury)

Les atomes sont composés d'un prédicat et d'une liste de termes appelés arguments. Comme nous l'avons vu préalablement dans la section 5.1.3, les arguments peuvent être des variables ou des valeurs. Dès lors les requêtes peuvent être soit composées d'un atome dont tous les arguments sont des valeurs soit composées d'un atome dont au moins un argument est une variable.

Dans le premier cas, les arguments sont uniquement composés de valeurs, par exemple, nous pourrions avoir une requête composée de l'atome suivant :

$$atom(estPereDe,\ [Pere1,\ Enfant2])$$

La réponse pour ce type de requête est booléenne, l'atome est donc soit vrai soit faux.

Dans le second cas, l'atome au cœur de la requête est composé d'au moins un argument étant une variable. L'atome est donc d'une forme similaire à celle qui suit :

$$atom(estPereDe,\ [Pere1,\ ?x])$$

ou encore :

$$atom(estPereDe,\ [?x,\ ?y])$$

Dans ces deux cas, la réponse n'est plus booléenne car nous souhaitons recevoir les valeurs pour lesquelles le prédicat est vérifié. La réponse du raisonneur sera donc un ensemble de substitutions de cette forme :

$$[(?x,\ Enfant1),\ (?x,\ Enfant2),\ (?x, Enfant3)]$$

Si cette réponse avait été donnée pour la première requête : $atom(estPereDe,\ [Pere1,\ ?x])$, cela signifierait que l'individu $Pere1$ est père des trois individus que sont : $Enfant1$, $Enfant2$ et $Enfant3$.

Lorsque le raisonneur évalue un atome, il doit trouver toutes les solutions possibles avant de continuer l'exécution. Ces solutions sont ensuite ajoutées à la cache sous la forme de *complete_value* au fur et à mesure qu'elles sont calculées.

Dans certains cas, des boucles infinies de résolution peuvent apparaître, celles-ci sont dues au fait que de nouveaux objectifs sont déjà des objectifs actuels. Cela engendre une boucle, par exemple, ces deux règles peuvent poser problème :

$$p1(?x,\ ?y) \wedge p2(?x, z) \Rightarrow p3(?x,\ ?y)$$

$$p3(?x,\ ?y) \wedge p4(?x,\ ?z) \Rightarrow p1(?z,\ ?y)$$

En effet, imaginions que le raisonneur reçoive la requête comprenant le prédicat $p3$, dès lors, en examinant la première règle, le prédicat $p1$ doit être évalué. Or la deuxième règle nous informe que pour que $p1$ soit évalué, $p3$ doit l'être car il fait partie du conséquent de la deuxième règle. Ceci engendre une boucle infinie étant donné que le raisonneur va à nouveau parcourir la première règle pour évaluer $p3$. Cet exemple est trivial et facilement détectable, cependant les règles peuvent être beaucoup plus éloignées et la détection de telles erreurs n'est pas tâche aisée.

Le raisonneur *backward*, lorsqu'il fait face à une boucle infinie, arrête son exécution et n'évalue donc pas les appels descendants de cette boucle. La solution qu'il retournera sera uniquement composée des valeurs déjà déduites pour l'atome de la requête.

En fin de résolution, le raisonneur marquera l'ensemble des solutions comme complet. Cela signifie que la prochaine fois qu'un appel similaire sera rencontré, les solutions devront simplement être consultée dans la cache. Cependant, le raisonneur n'est en mesure de marquer des solutions comme complètes que lorsque l'exécution n'a rencontré aucun événement critique tel qu'une boucle infinie.

4.3 Méthodologie

La méthodologie de ce travail est expérimentale, elle a pour but d'identifier les contraintes et les opportunités du raisonnement hybride.

La méthodologie mise en place peut être divisée en deux parties principales que sont l'implémentation d'un raisonneur *forward* et la mise en conjonction de ce raisonneur avec le raisonneur *backward* et son environnement décrits dans la section 4.2.

Le développement et la combinaison des deux raisonneurs se basent sur la partie II mais également sur l'expérimentation. Cette expérimentation permet d'explorer différentes possibilités afin d'en comparer les performances.

Nous parcourons donc dans le chapitre suivant l'ensemble des possibilités d'exécution entre les deux raisonneurs à savoir l'exécution parallèle et l'exécution séquentielle.

Chapitre 5

Solution

Partant du contexte décrit dans la section 4.2 schématisé dans la figure 4.1, il est désormais temps d'implémenter un raisonneur *forward* qui sera mené à interagir avec le raisonneur *backward*.

Le but final est donc d'obtenir deux raisonneurs interagissant parallèlement ou séquentiellement avec à la fois la cache, le programme mais également les faits.

L'objectif lié au contexte initial peut donc être schématisé par la figure 5.1 suivante.

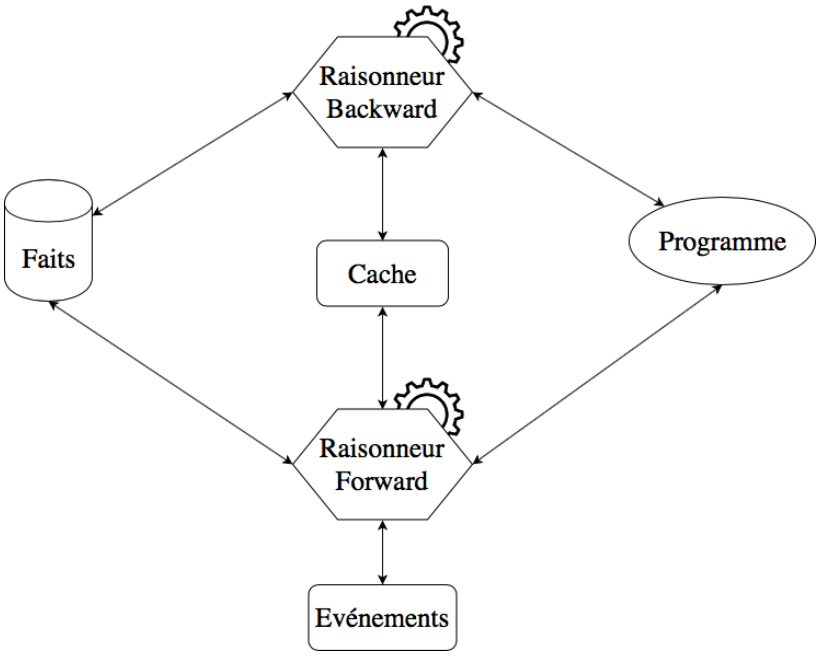

FIGURE 5.1 – Objectif d'intégration du raisonneur *forward* avec le raisonneur *backward*

5.1 Le raisonneur *forward chaining* et sa compatibilité avec le contexte initial

Nous l'avons déjà abordé dans la section 3.3, un raisonneur *forward* a besoin de deux éléments principaux, une base de connaissances initiale et un ensemble de règles. Ces deux éléments correspondent respectivement aux faits et au programme présents dans la figure 5.1. Cette figure associe cependant le raisonneur *forward* à deux autres éléments que sont la cache et les événements.

5.1.1 Les événements

Les événements représentent un sous ensemble des faits et sont les connaissances initiales sur lesquelles le raisonneur va être exécuté. En effet, il est rare d'exécuter un moteur d'inférence *forward* sur l'ensemble des données, une telle exécution s'avérerait extrêmement longue, onéreuse et, dans certains cas, inutile. Cette dernière serait inutile dans le sens où elle pourrait inférer un grand nombre de données inutiles au système. Cependant un raisonnement sur l'ensemble des données peut être cohérent dans le cas où un système est en cours d'initialisation ou de démarrage et aurait besoin d'inférer un maximum d'informations pour assurer un fonctionnement futur correct. Un événement est composé d'un prédicat et d'un ensemble de substitutions représentant les différentes valeurs pour lesquelles nous souhaitons inférer de nouvelles informations. Le raisonneur *forward* n'est évidemment pas exécuté sur base d'un seul prédicat, les événements de départ qui lui sont fournis sont donc une liste d'événements.

```
1 :- type forward_event
2     ---> event(
3         predicate :: swrl_predicate,
4         substs :: set(canon_subst)).
5
6 :- type events == list(forward_event).
```

Listing 5.1 – Type des événements *forward* (Mercury)

Au cours de l'exécution, également *depth-first*, cette liste d'événements est modifiée en permanence. Le raisonneur *forward* va sélectionner le premier événement de la liste, sélectionner les règles pertinentes [1] et inférer les données possibles à partir de ces dernières. Ces nouvelles données deviennent les événements de l'itération suivante. Lorsqu'un événement de la liste n'infère plus aucune connaissance, l'événement suivant est sélectionné. L'ordre de consultation des événements est de type *First-In-First-Out (FIFO)* [2].

1. Une règle est dite pertinente pour le *forward* si l'un des atomes présents dans l'antécédent contient le même prédicat que celui présent dans l'événement traité.

2. Le premier événement inféré (et donc ajouté à la liste d'événements) sera le premier à être traité.

5.1.2 La cache et les faits

La cache telle qu'utilisée par le raisonneur *backward* est tout à fait réutilisable par le moteur d'inférence *forward*. La description faite de cette dernière dans la section 4.2.3 est donc cohérente et adaptée à un raisonnement *forward*. Les assertions enregistrées par les deux types de raisonneurs peuvent donc être échangées et utilisées par les deux raisonneurs simultanément, aucune contrainte n'est identifiée à ce niveau.

Ceci est également valable pour les faits et leur représentation sous forme de triple RDF décrite dans la section 4.2.1. Cette représentation au sein du système est tout à fait exploitable et ne nécessite aucune modification de compatibilité.

5.1.3 Le programme

La programme est la représentation et la structure des règles. Ces dernières sont, dans le contexte décrit dans la section 4.2, optimisées pour un raisonnement *backward* ce qui n'est plus le cas dès à présent. Plusieurs éléments doivent être modifiés afin d'optimiser et faciliter le fonctionnement du raisonneur *backward*.

Nous abordons dans cette section les quatre modifications majeures apportées au programme afin d'améliorer sa compatibilité avec un raisonneur *forward*.

5.1.3.1 Indexation des règles

Les raisonneurs *backward* et *forward* ayant des fonctionnements opposés, l'un étant *top-down* et l'autre *bottom-up*, l'indexation des règles ne peut être identique.

Nous avons décrit dans la section 5.1.3 l'indexation propre au moteur *backward*. Cette dernière, indexant les règles par leur tête, n'a aucun intérêt dans le cadre d'un moteur d'inférence *forward*. En effet, un tel moteur juge de la pertinence des règles en fonction des prédicats présents dans leur antécédent. Une recherche par conséquent n'est donc d'aucun intérêt.

Une nouvelle représentation du programme a donc du être développée afin de fournir une indexation par le corps des règles et plus précisément par les prédicats présents dans la conjonction des antécédents.

Le listing 5.2 suivant décrit la *typeclass s2p_program* intégrant le prédicat *forward_applicable_rules* qui pour chaque programme et prédicat donnés en entrée retourne une *forward_mmap_body_heads* reprenant l'ensemble des règles pertinentes pour le prédicat en entrée. Vous pouvez remarquer qu'un autre paramètre est donné en entrée : *forward_param*, celui-ci reprend l'ensemble des *stores* disponibles pour le raisonneur. Nous verrons dans la section 5.2 que cette information est nécessaire lorsque les événements d'entrée sont restreints à certains ensembles. À partir de la ligne 23 du listing 5.2, seule la définition du prédicat *forward_applicable_rules* est repris, les définitions des autres prédicats et fonctions déclarés de la ligne 5 à 19 ne sont pas nécessaires dans le cadre de ce travail.

```
1  :- type forward_mmap_body_heads ==
       multi_map(swrl2prevail_body(s2p_pred_id),
       swrl2prevail_head(s2p_pred_id)).
2
3  %-----------------------------------------------------%
4
5  :- typeclass s2p_program(Prog) where [
6      func all_rules(Prog) = list(swrl2prevail_rule(s2p_pred_id)),
7
8      func applicable_rules(Prog, s2p_atom_pid) =
9          list(swrl2prevail_rule(s2p_pred_id)),
10
11     (pred forward_applicable_rules(Prog::in, swrl_predicate::in,
           forward_param::in,
12         forward_mmap_body_heads::out) is det),
13
14     (pred is_rule_var_like(Prog::in, swrl_var::in) is semidet),
15
16     (pred lookup_pred(Prog::in, s2p_pred_id::in,
           swrl2prevail_pred::out) is det),
17     (pred lookup_pred_rev(Prog::in, swrl2prevail_pred::in,
           s2p_pred_id::out) is det),
18     (pred lookup_pred_ids_for_uri(Prog::in, uri::in,
           set(s2p_pred_id)::out) is semidet)
19     ].
20
21 %-----------------------------------------------------%
22
23     :- instance s2p_program(mode_reordered_s2p_id_program) where [
24         (forward_applicable_rules(Prog, Pred, ForwardParam,
               MultiMap) :-
25             (ForwardParam ^ stores_available = [] ->
26                 (map.search(Prog ^ msrp_rules_per_body_pred ^
                       pred_body_head, Pred, MultiMap0) ->
27                     MultiMap = MultiMap0
28                 ;
29                     MultiMap = map.init
30                 )
31             ;
32                 AtomStores = Prog ^ msrp_rules_per_body_pred ^
                       atom_stores,
33                 ( map.search(Prog ^ msrp_rules_per_body_pred ^
                       pred_body_head, Pred, MultiMap0) ->
```

```
34                  map.to_assoc_list(MultiMap0, AssocListIn),
35                  list.filter_map((pred((Body - Heads)::in, (Body
                        - HeadListOut)::out) is semidet :-
36                      list.filter((pred(H::in) is semidet :-
37                          Atom = extract_atom_from_head(H),
38                          Atom = atom(Pred0, _),
39                          map.search(AtomStores, Pred0,
                                StoresNeeded),
40                          list.sublist(StoresNeeded, ForwardParam ^
                                stores_available)
41                          ),Heads, HeadListOut),
42                          not list.is_empty(HeadListOut)
43                      ), AssocListIn, AssocList),
44                  MultiMap = map.from_assoc_list(AssocList)
45              ;
46                  MultiMap = map.init
47              )
48          )
49      )
50  ].
```

Listing 5.2 – Représentation du programme pour le raisonneur *forward* et définition du prédicat de recherche de règle (Mercury)

La *forward_mmap_body_heads* est une multimap associant à chaque antécédent un ensemble de conséquents [3]. Le prédicat *forward_applicable_rules* recherche dans la multimap *msrp_rules_per_body_pred* contenue dans le programme les règles possédant le prédicat recherché dans l'antécédent. Cette multimap, comme vous l'aurez compris, associe à chaque prédicat l'ensemble des règles contenant ce prédicat dans l'antécédent.

5.1.3.2 Règles à conséquents multiples

Dans le cadre du raisonneur *backward*, la contrainte d'avoir des règles à tête unique semble évidente. En effet, grâce à cette contrainte, la recherche est plus aisée ainsi que l'indexation via les prédicats. Cependant dans le cadre du raisonnement *forward*, les têtes uniques peuvent s'avérer être un obstacle. En effet, imaginons deux règles dont les antécédents sont identiques mais dont les conséquents sont différents, dans le cadre d'un moteur *forward*, cela peut être synonyme de baisse de performance car les règles pourraient être traitées séparément. Les traiter séparément entraînera une résolution des antécédents redondantes étant donné qu'ils sont égaux.

3. Voir la section 5.1.3.2 pour la justification de cette représentation associant plusieurs têtes à un seul corps de règle.

La solution à ce problème réside dans le listing 5.2 qui, via le prédicat *forward_applicable_rules*, retourne une multimap associant à chaque antécédent tous les conséquents partageant ce même antécédent. Ceci permettra au raisonneur *forward* de résoudre l'antécédent une seule et unique fois tout en inférant les données présentes dans les différents conséquents.

Ce type de règles partageant le même antécédent ne sont pas rares, en effet, c'est notamment le cas de beaucoup de règles de sous-type. Les quatre règles suivantes, sans le mécanisme expliqué ci-dessus, auraient été traitées individuellement, l'antécédent *voiture(?x)* aurait donc été résolu quatre fois ce qui a un impact réel sur la performance.

$$voiture(?x) \Rightarrow vehicule(?x)$$
$$voiture(?x) \Rightarrow vehiculeTerreste(?x)$$
$$voiture(?x) \Rightarrow aQautreRoue(?x)$$
$$voiture(?x) \Rightarrow homologue(?x)$$

5.1.3.3 Assertions négatives

Le contexte initial basé sur les clauses de Horn et optimisé pour le raisonnement *backward* n'autorise pas les conséquents négatifs. Nous entendons par conséquent négatif le fait que la tête d'une règle puisse être composée à la fois d'atome comme nous l'avons vu dans le listing 4.2 mais également de complément d'atome. Le complément d'atome définit les individus qui ne répondent pas à un prédicat. Par exemple si l'individu $Jean$ est dans le complément d'atome dont le prédicat est $estUneFemme$ signifie que $Jean$ n'est pas une femme. Les compléments d'atome représentent donc bien la négation.

Le listing 5.3 propose désormais deux possibilités pour les têtes de règles à savoir les atomes (de façon similaire au listing 4.2) mais également les compléments d'atomes.

```
1  :- type rule(P,U,V)
2      ---> rule(
3               head :: rule_head(P,U,V),
4               body :: list(literal(P,U,V)),
5               name :: rule_name
6              ).
7
8   :- type rule_head(P, U, V)
9      ---> atom(atom(P, U, V))
10     ; complement_of_atom(atom(P, U, V)).
```

Listing 5.3 – Règle à tête négative (complément d'atome) (Mercury)

Lorsque le raisonneur *forward* infère de nouvelles assertions négatives, ces dernières sont également sauvées dans la cache. Cependant, elles ne font pas partie des événements qui sont, par définition, tous positifs. En effet, se baser sur des événements négatifs n'auraient aucun sens étant donné que les antécédents de règles sont uniquement composés d'atomes. Cependant, même si des assertions négatives faisaient partie des événements, celles-ci seraient ignorées car aucune règle pertinente ne serait trouvée.

5.1.3.4 *Parsing* négatif

Le concept de règle négative étant désormais défini et utilisable, générer des règles de ce type à partir des domaines définis par OWL et SWRL est le nouvel objectif.

Au sein d'une ontologie, plusieurs axiomes peuvent être traduits en règles à conséquent négatif :

— les classes disjointes,

— les propriétés disjointes,

— les propriétés asymétriques,

— les propriétés irréflexives.

Les classes disjointes sont exprimées dans OWL grâce au constructeur *owl :disjointWith*, ce constructeur garantit qu'un individu membre d'une classe ne peut pas être simultanément une instance d'une autre classe.

Imaginons le classe *Homme* sous-classe de *EtreVivant*, il est évident qu'un Homme n'est pas un animal ni un objet ni une plante. Cette classe serait donc définie comme ceci :

```
1 <owl:Class rdf:ID="Homme">
2   <rdfs:subClassOf rdf:resource="#EtreVivant"/>
3   <owl:disjointWith rdf:resource="#Animal"/>
4   <owl:disjointWith rdf:resource="#Objet"/>
5   <owl:disjointWith rdf:resource="#Plante"/>
6 </owl:Class>
```

Listing 5.4 – Classes disjointes (OWL)

Ce genre de classes, disjointes avec d'autres, peuvent donc engendrer de nouvelles règles à tête négative. Pour l'exemple du listing 5.4, les règles suivantes seraient générées :

```
1 rule(complement_of_atom(Animal(?x)), atom(Homme(?x)),
      "disjointHommeAnimal").
2 rule(complement_of_atom(Objet(?x)), atom(Homme(?x)),
      "disjointHommeObjet").
3 rule(complement_of_atom(Plante(?x)), atom(Homme(?x)),
      "disjointHommePlante").
```

Listing 5.5 – Règles pour classes disjointes (Mercury)

Le listing 5.5 utilise la nouvelle structure de règles définie dans la section 5.1.3.3.

Les propriétés disjointes sont similaires aux classes disjointes. Cependant ces disjonctions peuvent être de deux types étant donné qu'il existe deux types de propriétés, les propriétés d'objets et les propriétés de données.

Pour les propriétés d'objets, le constructeur *DisjointObjectProperties* est utilisé de la manière suivante dans OWL afin d'affirmer que deux propriétés d'objets sont disjointes :

```
1 DisjointObjectProperties( a:estPereDe a:estMereDe )
2 ObjectPropertyAssertion( a:estPereDe a:John a:Peter )
3 ObjectPropertyAssertion( a:estMereDe a:Maria a:Peter )
4
5 ObjectPropertyAssertion( a:estMereDe a:John a:Peter )
```

Listing 5.6 – Propriétés d'objets disjointes (OWL)

Les lignes 2 et 3 du listing 5.6 sont correctes étant donné que les individus sont bien différents. Cependant l'assertion à la ligne 5 n'est pas correcte étant donné que ces individus sont déjà utilisés pour l'assertion ligne 2 dont la propriété *estPereDe* est disjointe à la propriété *estMereDe*.

Les règles au sein du système seraient donc de la forme suivante :

```
1 rule(complement_of_atom(estMereDe(?x, ?y)), atom(estPereDe(?x,
      ?y)), "disjointPropMerePere").
2 rule(complement_of_atom(estPereDe(?x, ?y)), atom(estMereDe(?x,
      ?y)), "disjointPropPereMere").
```

Listing 5.7 – Propriétés d'objets disjointes au sein du système (Mercury)

Concernant les propriétés de données, seul le constructeur diffère, celui-ci est *DisjointDataProperties*. Le fonctionnement est semblable aux propriétés d'objets sauf que dans ce cas, les arguments des atomes peuvent être des données et non plus forcément de variables. Le constructeur *DisjointDataProperties* est utilisé de la manière suivante :

```
1 DisjointDataProperties( a:nom a:adresse )
2 DataPropertyAssertion( a:nom a:John "John Smith" )
3 DataPropertyAssertion( a:adresse a:Johen "Rue X, 999, 9999 XYZ" )
4
5 DataPropertyAssertion( a:adresse a:John "John Smith" )
```

Listing 5.8 – Propriétés de données disjointes (OWL)

Le listing 5.8 affirme, à la ligne 1, que les propriétés d'adresse et de nom sont disjointes. Dès lors la ligne 5 n'est pas correcte étant donné que la valeur *"John Smith"* a déjà été assignée au nom de l'individu John.

Les propriétés asymétriques sont l'inverse des propriétés symétriques abordées dans la section 2.5.2.3 et la définition 5.2. Les propriétés asymétriques sont obligatoirement des propriétés d'objet, l'axiome utilisé pour les définir est *AsymmetricObjectProperty*.

Définition 5.1. (AsymmetricProperty). Si une propriété, P, est asymétrique alors :

$$\forall x \text{ et } y : P(x,\ y) \text{ si et seulement si } \neg P(y,\ x)$$

Les propriétés asymétriques sont définies dans OWL de façon similaire au listing 5.9 suivant :

```
1 AsymmetricObjectProperty( a:estPereDe )
2 ObjectPropertyAssertion( a:estPereDe a:Peter a:John )
3
4 ObjectPropertyAssertion( a:estPereDe a:John a:Peter )
```

Listing 5.9 – Propriétés d'objets asymétriques (OWL)

Étant donné que la propriété *estPereDe* est asymétrique, définir la ligne 4 du listing 5.9 n'est pas valide étant donné que l'individu *Peter* a été défini comme étant le père de l'individu *John* à la ligne 2.

Une telle propriété asymétrique aurait la représentation suivante au sein du programme :

```
1 rule(complement_of_atom(estPereDe(?y, ?x)), atom(estPereDe(?x,
      ?y)), "asymmetricPropPere").
```

Listing 5.10 – Propriété d'objet asymétrique (Mercury)

Le dernier type de propriété pouvant être interprété par des règles à conséquent négatif sont les propriétés irréflexives. Similairement aux propriétés asymétriques, les propriétés irréflexvies sont uniquement des propriétés d'objet. La définition d'une propriété irréflexive est la suivante :

Définition 5.2. (IrreflexiveProperty). Si une propriété, P, est irréflexive alors :

$$\forall x\ : \neg P(x,\ x)$$

Les propriétés irréflexives sont déinies grâce à l'axiome *IrreflexiveObjectProperty* dans OWL, le listing en est un exemple d'utilisation.

```
1 IrreflexiveObjectProperty( a:estPereDe )
2
3 ObjectPropertyAssertion( a:estPereDe a:Peter a:Peter )
```

Listing 5.11 – Propriétés d'objets irréflexives (OWL)

Le listing 5.11 définit donc la propriété *estPereDe* comme étant irréflexive, l'individu *Peter* ne peut donc pas être son propre père.

Une telle propriété engendrerait les règles suivantes au sein du système :

```
1 rule(complement_of_atom(estPereDe(?x, ?x)), atom(estPereDe(?x,
      ?y)), "asymmetricPropPere").
2 rule(complement_of_atom(estPereDe(?y, ?y)), atom(estPereDe(?x,
      ?y)), "asymmetricPropPere").
```

Listing 5.12 – Propriété d'objet irréflexive (Mercury)

5.1.4 La résolution d'antécédent (conjonction)

Comme nous l'avons décrit, le raisonneur *forward* est *depth-first*, il va donc sélectionner un événement et sélectionner, pour celui-ci, les règles pertinentes comme nous l'avons décrit dans la section 5.1.1. Ces règles seront regroupées en fonction de leur antécédent afin d'éviter la redondance de résolution (cf. la section 5.1.3.2).

Les antécédents seront résolus les uns après les autres afin d'inférer les assertions présentes dans les différents conséquents associés à chaque corps de règle.

La résolution d'antécédent repose sur les substitutions. A chaque résolution de corps de règle est associé une substitution. Pour rappel, un corps est sélectionné si le prédicat présent dans l'événement est présent dans l'un des atomes présents dans la conjonction qui compose l'antécédent de la règle. Imaginons un antécédent de règle composé des atomes suivante :

$$atom(estPereDe(?y, ?x)),\ atom(estFrereDe(?x, ?z))$$

Et que l'événement traité actuellement est le prédicat *estPereDe* avec les individus *"Peter"* et *"John"* :

$$estPereDe(\text{"}Peter\text{"}, \text{"}John\text{"})$$

Avant de commencer la résolution de l'antécédent, une substitution initiale va être initialisée afin d'associer les individus déjà connus (*Peter* et *John*) aux variables présentes dans l'antécédent (*?y* et *?x*). La substitution aura donc la forme suivante :

$$(?x \rightarrow \text{"}John\text{"}),\ (?y \rightarrow \text{"}Peter\text{"})$$

À partir de cette substitution, la raisonneur va, pour les autres atomes présents dans l'antécédent, remplacer les variables par les valeurs présentes dans la substitution. Dès lors l'atome : $atom(estFrereDe(?x, ?z))$ devient $atom(estFrereDe(\text{"}John\text{"}, ?z))$. Le raisonneur est dès lors en mesure de résoudre cette atome, c'est-à-dire rechercher les valeurs de la variable $?z$ pour lesquelles l'atome $atom(estFrereDe(\text{"}John\text{"}, ?z))$ est vérifié. Dès lors, trois possibilités se profilent :

— une valeur est trouvée pour la variable $?z$. Cette valeur associée à la variable est ajoutée à la substitution initiale. Imaginons que la valeur trouvée soit "*Leonard*", la substitution deviendra donc :

$$(?x \rightarrow \text{"}John\text{"}),\ (?y \rightarrow \text{"}Peter\text{"}),\ (?z \rightarrow \text{"}Leonard\text{"})$$

— plusieurs valeurs sont trouvées pour la variable $?z$. Pour chaque valeur, la substitution initiale est copiée et chaque valeur associée à la variable est ajoutée à une substitution différente. Imaginons que les valeurs trouvées soient "*Leonard*" et "*Bradley*", il y aura donc deux substitutions différentes à savoir :

$$(?x \rightarrow \text{"}John\text{"}),\ (?y \rightarrow \text{"}Peter\text{"}),\ (?z \rightarrow \text{"}Leonard\text{"})$$

$$(?x \rightarrow \text{"}John\text{"}),\ (?y \rightarrow \text{"}Peter\text{"}),\ (?z \rightarrow \text{"}Bradley\text{"})$$

— aucune valeur n'est trouvée pour la variable $?z$. Si aucune valeur n'est trouvée, ni dans la cache ni dans les faits alors, la résolution échoue et aucune assertion ne pourra être inférée à partir de cet antécédent.

Le raisonneur poursuivra la résolution de l'antécédent en appliquant à nouveau le procédé décrit ci-dessus.

Une fois la résolution de l'antécédent et la ou les substitution(s) établies, il est désormais possible d'inférer les assertions des différents conséquents associés au corps résolu. Les substitutions sont donc appliquées aux atomes présents dans les différentes têtes et les assertions sont ajoutées à la cache. Cependant ces assertions ne peuvent pas être enregistrées sous forme de *complete_value* étant donné que le raisonneur *forward* n'est pas en mesure d'affirmer que ces informations sont complètes.

Le listing 5.13 suivant correspond au code pour la résolution d'un antécédent pour un événement donné.

```
1 resolve_body_event_subst(Prog, KB, Body0, HeadsIn, Pred,
      CanonSubst, NewPositiveEvents, !HeadsToSetComplete, !Cache) :-
2     list.map(lookup_pred_in_literal(Prog), Body0, Body),
3     generate_substs_for_applicable_atoms(Pred, CanonSubst, Body,
          StartSubsts0),
```

```
4       (!.Cache ^ parent_cache = yes(Parent) ->
5           ParentCache = Parent
6       ;
7           error("The Cache must have a parent_cache")
8       ),
9       list.filter((pred(H::in) is semidet :-
10          Atom = extract_atom_from_head(H),
11          lookup_pred_in_atom(Prog, Atom, AtomWithPred),
12          AtomWithPred ^ predicate ^ s2p_pred_type_and_name =
                dlInferredPredicate(SwrlPred),
13          (SwrlPred = owl_class(owl_Thing) ->
14              false
15          ;
16              set.filter((pred(Subst::in) is semidet :-
17                  list.map(pred(Argument::in,
                        ArgumentSubstituted::out) is det :-
18                      substitute_term(Subst, Argument,
                            ArgumentSubstituted), Atom ^ arguments,
                            Arguments),
19                  list.filter(is_term_value, Arguments, ArgumentsOut),
20                  list.length(ArgumentsOut, L),
21                  list.length(Atom ^ arguments, L1),
22                  (L \= L1 ->
23                      true
24                  ;
25                      not search(ParentCache ^ answers, atom(Atom ^
                            predicate, ArgumentsOut), _)
26                  )
27                  ), StartSubsts0, StartSubsts0Out),
28
29              set.count(StartSubsts0Out) \= 0
30              )
31          )
32          ,HeadsIn, Heads),
33      (Heads = [] ->
34          NewPositiveEvents = []
35      ;
36          trace [compile_time(flag("reasoner_trace")),
                run_time(env("REASONER_TRACE")), io(!IO)]
37              trace_event(kb_get_reasoner_id(KB), extend(Body,
                    CanonSubst), !IO),
38          set.filter(pred(S::in) is semidet :- not
                is_empty_substitution(S), StartSubsts0, StartSubsts),
39          (StartSubsts = set.init ->
```

```
40              NewPositiveEvents = [],
41              NewNegativeEvents= []
42          ;
43              set.map_fold(pred(StartSubst::in, SetUniSubs::out,
                    CacheIn::in, CacheOut::out) is det :-
44                  (BodySubstituted =
                        list.map(substitute_literal(StartSubst),
                        Body),
45                  prevail_priscus_literals_conjunction_query(KB,
                        fb_prog(Prog), BodySubstituted,
                        SetSubstsOut, CacheIn, CacheOut),
46                  set.filter_map(subst_union(StartSubst),
                        SetSubstsOut, SetUniSubs)),
47                StartSubsts, SetSetSubsts, !Cache),
48              SetSubsts = set.list_to_set(list.condense(
49                  list.map(set.to_sorted_list,
50                  set.to_sorted_list(SetSetSubsts))
51                  )),
52
53              (!.Cache ^ parent_cache = yes(ParentCache) ->
54                  generate_events_and_save(Prog, Heads, SetSubsts,
                        NewPositiveEvents, NewNegativeEvents,
                        !HeadsToSetComplete, ParentCache,
                        ParentCacheOut),
55                  !Cache ^ parent_cache := yes(ParentCacheOut)
56              ;
57                  error("must have parent")
58              )
59          ),
60          trace [compile_time(flag("reasoner_trace")),
                run_time(env("REASONER_TRACE")), io(!IO)]
61              (trace_event(kb_get_reasoner_id(KB), end_extend(Body,
                    CanonSubst, NewPositiveEvents, NewNegativeEvents),
                    !IO))
62      ).
```

Listing 5.13 – Résolution d'un antécédent (Mercury)

5.2 Intégration des raisonneurs *forward* et *backward*

L'intégration et la mise en corrélation du raisonneur *forward* décrit ci-dessus avec un raisonneur *backward* soulève plusieurs questions quant à l'organisation des exécutions des différents raisonneurs. L'organisation retenue est abordée dans la sec-

tion5.2.1. Dans un second temps nous abordons le type de raisonnement *forward* à choisir afin d'optimiser les performances du raisonneur hybride.

5.2.1 Organisation des exécutions des deux raisonneurs

Bien que l'agencement des exécutions des deux raisonneurs soit une question importante, le fait que le raisonneur *backward* soit le raisonneur principal n'en est pas une. En effet les requêtes sont toujours adressées à ce dernier, nous pouvons donc voir le raisonneur *forward* comme une aide à ce raisonneur principal, il tentera de déduire des assertions utiles au *backward* sans pour autant inférer trop d'assertions inutiles.

Deux choix s'offrent à nous quant à l'organisation des raisonneurs :

— l'exécution parallèle des deux raisonneurs.

— l'exécution préalable du raisonneur *forward* à celle du raisonneur *backward*.

5.2.1.1 Exécution parallèle des raisonneurs

L'exécution parallèle paraît, au premier abord une solution de choix. En effet, les deux raisonneurs exécutés en même temps permettraient que leurs exécutions se rejoignent en un point précis. La figure suivante représente ce type de raisonnement hybride, comme vous pouvez le constater, les raisonneurs évolueraient simultanément jusqu'à ce que le raisonneur *backward*, essayant de résoudre la requête émise, utilise les assertions inférées par le raisonneur *forward* et non plus sur les connaissances initiales.

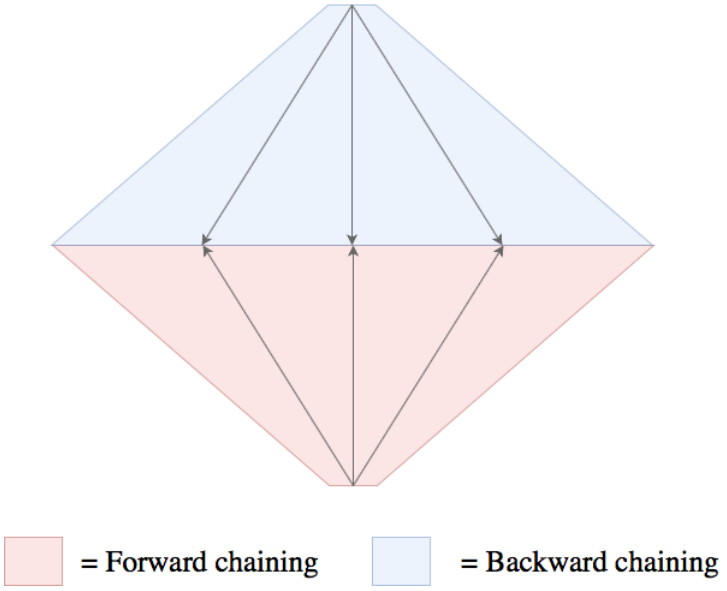

FIGURE 5.2 – Raisonnement hybride avec exécution parallèle

Malheureusement, cette idée bien qu'elle semble être idéal, pose problème. En effet, bien que le fonctionnement du raisonneur *forward* soit correcte, ses résultats

ne peuvent pas être exploités par le raisonneur *backward*. Ceci est du au fait que le raisonneur *forward* n'est pas en mesure de sauver dans la cache des informations complètes [4]. Étant donné que les informations sont incomplètes, le raisonneur *backward* les ignore et poursuit son exécution classique afin d'obtenir une résolution complète. Le raisonneur *backward* va donc parcourir toutes les règles qu'il aurait parcourues si aucune information n'était présente dans la cache parce qu'il n'est pas en mesure de savoir quelles règles ont déjà été parcourues par le raisonneur *forward*.

5.2.1.2 Exécution séquentielle des raisonneurs

Ayant éliminé l'exécution parallèle, la question ne se pose plus, l'exécution se fera séquentiellement. Le raisonneur *forward* aura une exécution préalable de celle du raisonneur *backward*.

Cette exécution séquentielle est représentée par la figure suivante :

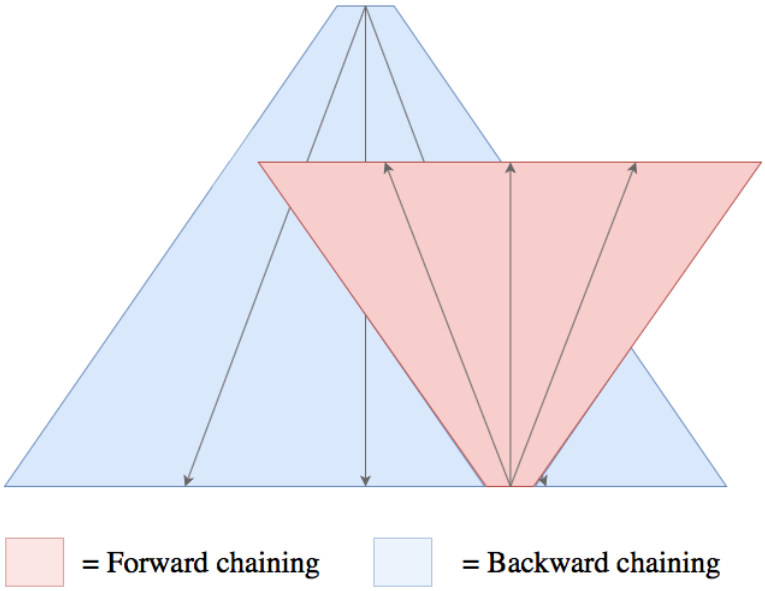

FIGURE 5.3 – Raisonnement hybride avec exécution séquentielle

L'exécution séquentielle permet au raisonneur *forward* de terminer son exécution avant que le raisonneur *backward* n'utilise ses assertions. Ayant terminé son exécution, celui-ci est capable d'affirmer que l'ensemble des assertions sont complètes car il aura parcouru toutes les variantes possibles.

L'enjeu de ce raisonnement séquentielle est de ne pas avoir un ensemble d'événements trop important à fournir au raisonneur *forward* dans le but de limiter son rayon d'assertions et de ne donc pas inférer trop d'informations inutiles. Nous traitons de ce sujet dans la section 5.2.2 suivante.

4. *complete_value*, voir la section 4.2.3

Le flux d'informations est donc semblable à ce lui de Jena abordé dans la section 3.5.2.1 et schématisé par la figure 3.9. La figure 5.4 ci-dessous en est l'illustration.

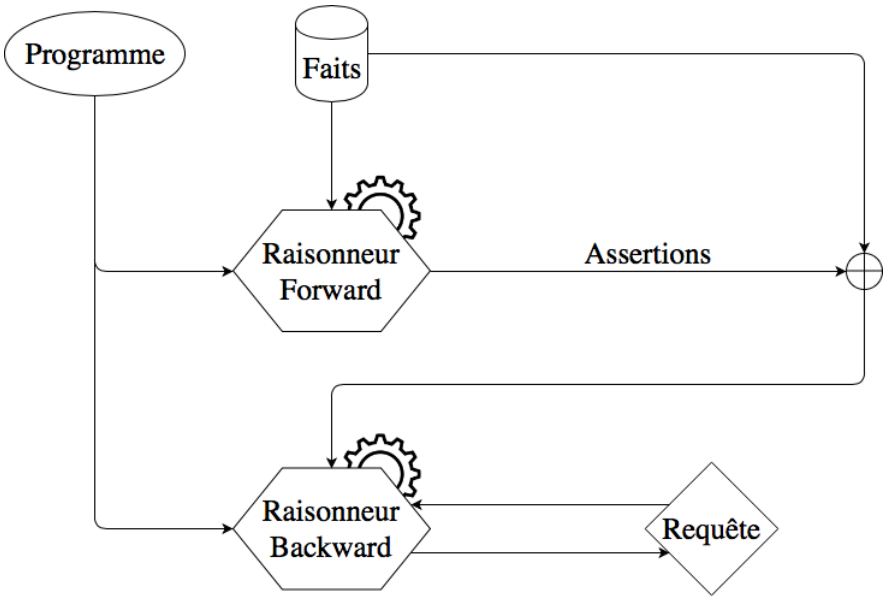

FIGURE 5.4 – Flux d'information du raisonneur hybride

5.2.2 Quel type de raisonnement *forward*

Les exécutions désormais définies comme séquentielles, la problématique des assertions inutiles reste d'actualité. L'idée est d'éviter la situation illustrée dans la figure suivante, des assertions inférées préalablement par le raisonneur *forward* mais dont le raisonneur *backward* ne tire pas profit. Les assertions inutiles sont située dans les deux triangles externes de couleur jaune.

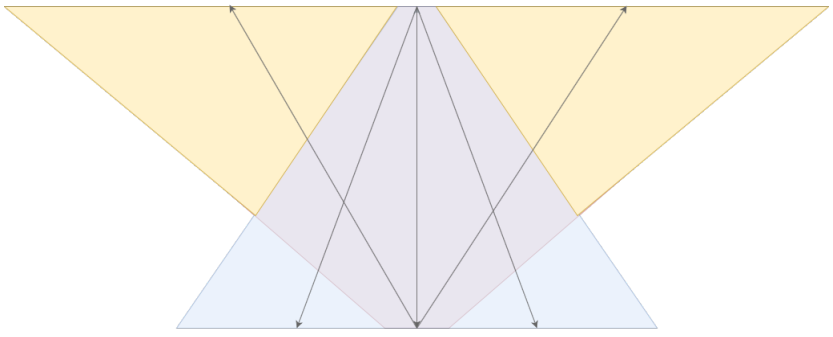

FIGURE 5.5 – Raisonneur *forward* avec trop d'assertions inutiles

Afin d'éviter ce problème, deux solutions potentielles existent :

— limiter les événements fournis au raisonneur *forward* avant son exécution,

— limiter les règles auxquelles le raisonneur *forward* a accès.

Nous abordons et comparons ces deux solutions dans la section 5.2.2.3 suivante.

5.2.2.1 Limiter les événements initiaux

Cette technique consiste, lorsqu'une requête est émise au raisonneur *backward*, à fournir des événements précis et restreints au raisonneur *forward*.

Comme nous l'avons décrit dans la section 5.1.3.2, le raisonneur *forward* est fortement approprié pour les règles à antécédent identique. Les règles énoncées avec l'antécédent $voiture(?x)$ abordées plus dans la section 5.1.3.2 en sont un exemple, lorsque le raisonneur résout le conséquent, il ne le fera qu'une seule fois pour inférer l'ensemble des conséquents. Les règles partageant un même antécédent sont souvent des règles liées au type étant donné qu'il existe beaucoup de règles dont le corps est simplement une vérification de type.

Dès lors, l'idée est, avant que le raisonneur *backward* n'évalue la requête propre à un ou plusieurs individus, de fournir comme événements au moteur *forward* l'ensemble des assertions relatives aux types de ces individus.

Bien que cette solution s'avère efficace, son utilisation peut poser problème lorsqu'il s'agit de la complétude des données sauvegardées dans la cache. Une solution s'offre à nous afin de permettre d'affirmer que les données sont complètes sans pour autant avoir accès à l'intégralité des données initiales en tant qu'événement : trier les règles afin de ne retenir que celles dont tous les atomes composant l'antécédent sont connus comme étant complets.

Cette solution paraît à premier abord être adéquate, cependant après plusieurs essais sur différentes ontologies [5], cette solution ne se révèle pas optimale. La condition émise sur la sélection des règles est trop restrictive. Les règles dont les atomes présents dans l'antécédent sont connus comme complets sont rares, seule une minorité de règles est donc sélectionnée. Les performances d'inférence du raisonneur *forward* sont donc fortement réduites voir inexistantes.

Cette solution n'est donc pas utilisée et les atomes sont sauvegardés de façon incomplète de la façon suivante :

$$atom(Homme(?x)),\ incomplete\_value([(?x \rightarrow "John"),\ (?x \rightarrow "Peter")])$$

Ceci ne veut pas dire qu'ils seront ignorés par le raisonneur *backward*, en effet, si la requête spécifie les individus et donc qu'une variable n'est présente dans la requête, la cache sera en mesure d'affirmer que la requête est valide. Imaginons que la requête soit composée de l'atome suivant :

$$atom(Homme("Peter"))$$

5. Les ontologies testées ont été fournies par ODASE Ontologies, elles ne peuvent donc pas figurer dans ce texte.

La cache sera capable d'affirmer que cette requête est vraie car *Peter* est bien présent dans la substitution bien qu'elle ne soit pas complète. Cependant, si *Peter* n'était pas présent dans les substitutions, alors on ne pourrait ni affirmer que *Peter* est un Homme ou qu'il n'en est pas un. Le raisonneur *backward* doit donc continuer son exécution pour l'affirmer.

5.2.2.2 Limiter les règles accessibles

La deuxième possibilité pour réduire les assertions inutiles est de limiter l'ensemble des règles que le moteur *forward* a à sa disposition. Nous avons préalablement abordé, dans le listing 5.2, le prédicat *forward_applicable_rules*. Ce dernier prend un paramètre appelé *ForwardParam*, ce dernier contient l'attribut *stores_available* étant une liste des *stores* disponibles pour le raisonneur. Ces *stores* sont, en fonction de la requête émise au raisonneur *backward*, sélectionnés et transmis au moteur *forward*.

Le raisonneur *forward* est donc appelé avec l'identifiant des *stores* qu'il a à sa disposition, ce dernier va en extraire l'ensemble des faits y étant stockés et les transformer en événements afin de commencer son exécution. Lorsque ce dernier devra sélectionner les règles pertinentes pour l'événement qu'il est en train de résoudre, celles-ci seront sélectionnées si et seulement si :

— elles sont pertinentes. Ce qui signifie que le prédicat présent dans l'événement est présent dans l'un des atomes de l'antécédent de cette règle.

— le résultat (les conséquents) peut être affirmer comme complet lorsqu'il aura été inférer. Le résultat d'une règle sera complet si l'ensemble des *stores* utilisés pour stocker les prédicats présents dans l'antécédent de cette règle est un sous-ensemble des *stores* passés en paramètre au raisonneur *forward*.

5.2.2.3 Comparaison des deux possibilités

Les deux graphiques ci-dessous ont pour but de comparer les deux possibilités de solutions détaillées ci-dessus. Ces tests ont été réalisés sur deux modèles ontologiques différents, l'un issu du monde bancaire l'autre du monde militaire.

Les tests consistent en une requête *GET* émise à un service Web ayant pour but d'inférer l'ensemble des assertions possibles à partir d'une ressource passée en paramètre.

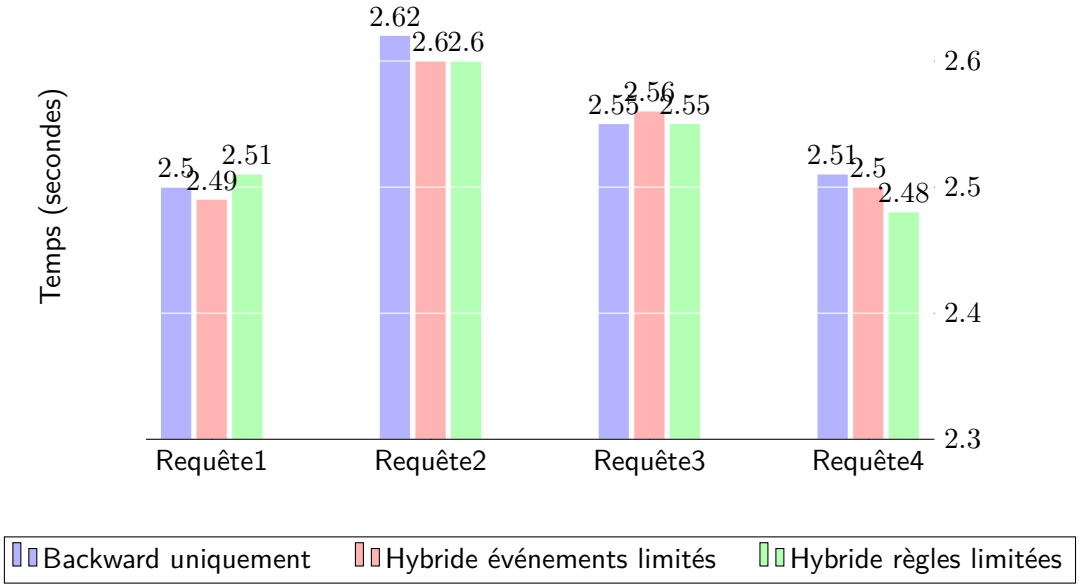

FIGURE 5.6 – Comparaison des temps d'exécution de la requête (modèle militaire)

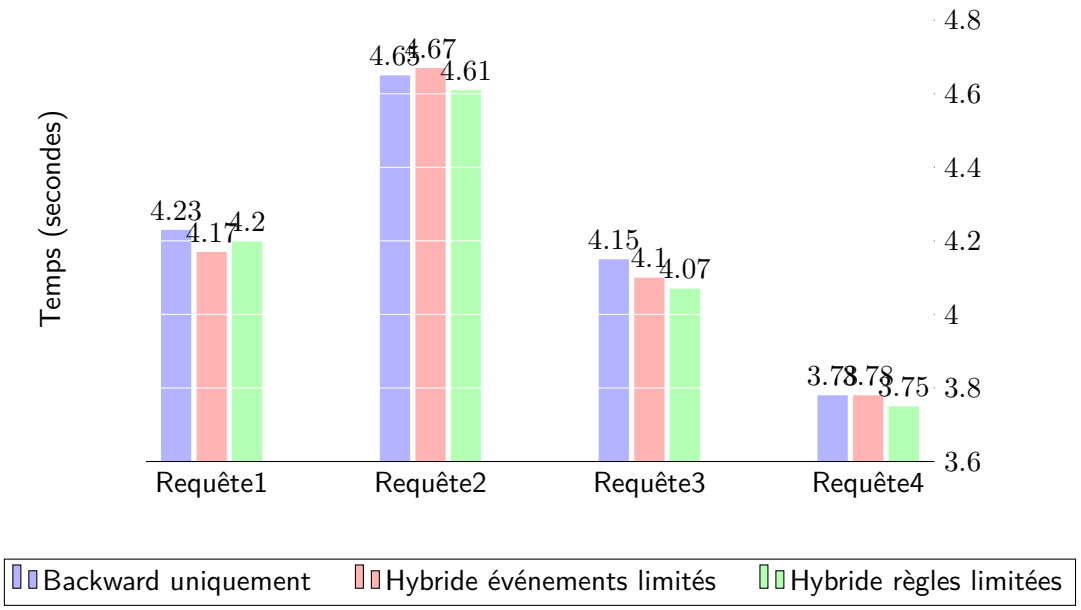

FIGURE 5.7 – Comparaison des temps d'exécution de la requête (modèle bancaire)

Les deux ontologies testées ont été modélisées avant la réalisation de ce travail, certaines bonnes pratiques *backward* telles qu'une attention particulière à éliminer les boucles infinies ont été utilisées afin d'optimiser les performances de ce raisonneur *backward*. Cependant, l'utilisation des deux possibilités de solution que sont la réduction des règles à utiliser et la limitation des événements initiaux, ont permis d'obtenir

des temps de réponse aux requêtes en moyenne légèrement inférieures aux temps de réponse du raisonneur *backward* seul. Bien que la différence entre les temps d'exécution ne soit pas aussi importante que nous l'espérions, elles ne sont néanmoins pas inintéressantes étant donné que les modèles sont optimisés au raisonnement *backward*. Obtenir un gain, aussi minime soit-il est un résultat positif.

Un gain, en utilisant le raisonnement hybride, n'est potentiellement possible que lorsque que le raisonneur *backward* utilise à de multiples reprises les assertions inférées préalablement par le raisonneur *forward*. En 1985, l'article [27] définissait plusieurs méthodes d'implémentation de programmes logiques. Parmis celles-ci figure les *Magic Sets* étant un algorithme général de réécriture des règles logiques afin qu'elles puissent être implémentées par le un moteur d'inférence *forward*. Cet article introduisant plusieurs méthodes qualifiée d'étranges, a pour but d'introduire les bonnes méthodes de résolutions de requêtes mais également de mettre en avant la difficulté de prouver quoi que ce soit quant aux moyens optimaux de résolutions de requêtes.

Quatrième partie

Travaux futurs et conclusion

Chapitre 6

Travaux futurs

Plusieurs travaux et expériences peuvent encore être réalisés afin de préciser et d'augmenter les performances du raisonnement hybride.

6.1 Contexte de test différent

Premièrement, les tests réalisés dans la section 5.2.2.3 sur des modèles ontologiques modélisés selon des normes *backward* ne s'avèrent pas représentatifs. Tester les deux différents type de raisonnement hybride sur des ontologies modélisées sans prendre en compte ces bonnes pratiques *backward* permettrait d'obtenir des résultats potentiellement différents.

6.2 Test sur le nombre d'inférences

Les tests réalisés dans la section 5.2.2.3 ne tiennent compte que du temps d'exécution des requêtes. Cependant, afin d'identifier précisément d'où ce gain émerge, un compteur d'inférence serait utile. Grâce à ce dernier, nous pourrions identifier quelles requêtes ont pu être simplifiées grâce aux assertions inférées préalablement par le raisonneur *forward*.

6.3 Gestion du raisonneur *forward* par le *backward*

Deux organisations d'exécution ont été parcourues à savoir l'exécution parallèle et l'exécution séquentielle. Un autre type d'exécution pourrait être imaginé, l'exécution imbriquée. Il s'agirait d'appeler classiquement le raisonneur *backward* avec une requête, ce dernier, lorsqu'il détecterait des tâches d'inférence complexes telles que l'utilisation de règles avec des conséquents identiques serait capable d'appeler le moteur *forward* pour résoudre ces tâches peu adaptées à son type de raisonnement.

Le raisonneur *backward* serait donc le raisonneur principal pouvant faire appel au raisonneur *forward* à tout moment propice de son exécution. Le moteur *backward*

serait alors en pause lorsque le raisonneur *forward* infère les assertions via des règles qui lui sont adaptées.

6.4 Utilisation du raisonneur *forward* pour des tâches de type *start-up*

Un dernier potentielle horizon de recherche serait l'utilisation du raisonneur *forward* dès le *start-up*. Dès l'accès initial au système, lorsque aucune requête n'est encore émise au raisonneur *backward*, le moteur *forward* pourrait être exécuté afin d'inférer des assertions de base permettant de faciliter de futures résolutions de requête.

Chapitre 7

Conclusion

Le but de ce mémoire est, dans un premier temps, de comprendre le fonctionnement du Web sémantique en explorant ce dernier couche après couche. Chaque couche est liée à ses propres technologies qu'il est nécessaire de maîtriser pour, dans un second temps, comprendre les différents types de raisonnement reposant sur celles-ci.

Malheureusement peu d'applications utilisant le raisonnement hybride existent à l'heure actuelle, un système cependant en propose une application concrète : Apache Jena. Cette application concrète se base sur une exécution séquentielle des raisonneurs, le moteur *forward* est en effet exécuté avant le moteur *backward*. Ce type d'exécution séquentielle a été testée dans ce travail et s'avère être plus performante et adaptée qu'une exécution parallèle que plusieurs facteurs rendent inefficaces. Ces facteurs sont principalement liés à la complétude des données inférées.

Notre raisonneur hybride ayant étant défini comme séquentiel, éviter les assertions inutiles est le second problème auquel nous faisons face. Deux possibilité étaient envisageables : limiter les règles auxquelles le raisonneur *forward* a accès ou limiter les événements de départ. Dans ce cas, les tests ont prouvés que les deux possibilités étaient relativement identiques et n'impactaient pas grandement les temps de résolution de requête.

En conclusion, il ressort de ce travail, réalisé dans un contexte précis, que le raisonnement hybride obtient de plus grande performance lorsqu'il est séquentiel. Cependant, les futurs travaux pourraient déterminer qu'une exécution imbriquée puisse également être cohérente.

Bibliographie

[1] Tim Berners-Lee, James Hendler et Ora Lassila. « THE SEMANTIC WEB ». Dans : *Scientific American* 284.5 (2001), p. 34–43. ISSN : 0036-8733.

[2] *World Wide Web Consortium*. 2018. URL : https://www.w3.org.

[3] *Resource Description Framework*. 2014. URL : https://www.w3.org/RDF/.

[4] *RDF Schema*. 2014. URL : https://www.w3.org/TR/rdf-schema/.

[5] *Web Ontology Language*. 2012. URL : https://www.w3.org/OWL/.

[6] Philippe Laublet, Chantal Reynaud et Jean Charlet. *Sur quelques aspects du Web sémantique*. 2002.

[7] Grigoris Antoniou et Frank Van Harmelen. *A Semantic Web Primer*. Google-Books-ID : cIYAiS1wbRsC. MIT Press, 2004. ISBN : 978-0-262-01210-2.

[8] *Extensible Markup Language (XML)*. 2016. URL : https://www.w3.org/XML/.

[9] Mike Uschold et Michael Gruninger. « Ontologies : Principles, Methods and Applications ». Dans : *Knowledge Engineering Review* 11 (1996), p. 93–136.

[10] Roberto Poli. « Ontology for Knowledge Organization ». Dans : *Knowledge organization and change* (1996).

[11] R. Van de Riet, H. Burg et F. Dehne. « Linguistic Issues in Information Systems Design ». Dans : *Formal Ontology in Information Systems*. Sous la dir. de NicolaEditor Guarino. Ios Press, 1998.

[12] Maurizio Vincini et Silvana Castano. *S. Bergamaschi Università degli Studi di Modena e Reggio Emilia*.

[13] Ewald Lang. « The LILOG ontology from a linguistic point of view ». Dans : *Text Understanding in LILOG*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1991, p. 464–481. ISBN : 978-3-540-54594-1. DOI : 10.1007/3-540-54594-8_76. URL : https://link.springer.com/chapter/10.1007/3-540-54594-8_76.

[14] Dieter Fensel. « Ontologies ». Dans : *Ontologies*. Springer, Berlin, Heidelberg, 2001. ISBN : 978-3-662-04398-1. DOI : 10.1007/978-3-662-04396-7_2. URL : https://link.springer.com/chapter/10.1007/978-3-662-04396-7_2.

[15] *URI*. URL : https://www.w3.org/wiki/URI.

[16] *URIs, URLs, and URNs : Clarifications and Recommendations 1.0*. URL : https://www.w3.org/TR/uri-clarification/.

[17] *DAML+OIL (March 2001) Reference Description*. 2001. URL : https://www.w3.org/TR/daml+oil-reference.

[18] Ritesh AGRAWAL. *Difference between OWL Lite, DL, and Full*. Fév. 2007. URL : https://ragrawal.wordpress.com/2007/02/20/difference-between-owl-lite-dl-and-full/.

[19] *SWRL : A Semantic Web Rule Language Combining OWL and RuleML*. 2004. URL : https://www.w3.org/Submission/SWRL/.

[20] Boris MOTIK et al. « Representing ontologies using description logics, description graphs, and rules ». Dans : *Artificial Intelligence* 173.14 (sept. 2009), p. 1275–1309. ISSN : 00043702. DOI : 10.1016/j.artint.2009.06.003.

[21] Kazunori UEDA. « Guarded horn clauses ». Dans : *Logic Programming '85*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, juil. 1985, p. 168–179. ISBN : 978-3-540-16479-1. DOI : 10.1007/3-540-16479-0_17. URL : https://link.springer.com/chapter/10.1007/3-540-16479-0_17.

[22] *Apache Jena - Reasoners and rule engines : Jena inference support*. URL : https://jena.apache.org/documentation/inference.

[23] *Pellet : A practical OWL-DL reasoner - ScienceDirect*. URL : https://www.sciencedirect.com/science/article/pii/S1570826807000169.

[24] Minsu JANG et Joo-Chan SOHN. « Bossam : An Extended Rule Engine for OWL Inferencing ». Dans : *Rules and Rule Markup Languages for the Semantic Web*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, nov. 2004, p. 128–138. ISBN : 978-3-540-23842-3. DOI : 10.1007/978-3-540-30504-0_10. URL : https://link.springer.com/chapter/10.1007/978-3-540-30504-0_10.

[25] *The Mercury Project*. URL : https://www.mercurylang.org/.

[26] Terrance SWIFT et David S. WARREN. « An Abstract Machine for SLG Resolution : Definite Programs ». Dans : *In Proceedings of the Symposium on Logic Programming*. 1994, p. 633–654.

[27] Francois BANCILHON et al. « Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract) ». Dans : *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. PODS '86. ACM, 1986, p. 1–15. ISBN : 978-0-89791-179-5. DOI : 10.1145/6012.15399. URL : http://doi.acm.org/10.1145/6012.15399.